Digital Technical Journal

digital

PARALLEL SCSI TECHNOLOGY

ROUTER CLUSTERS

MULTIPLATFORM 3-D APPLICATION SHARING

HPF DEBUGGER

configuration expansion

speed change/hot plugging

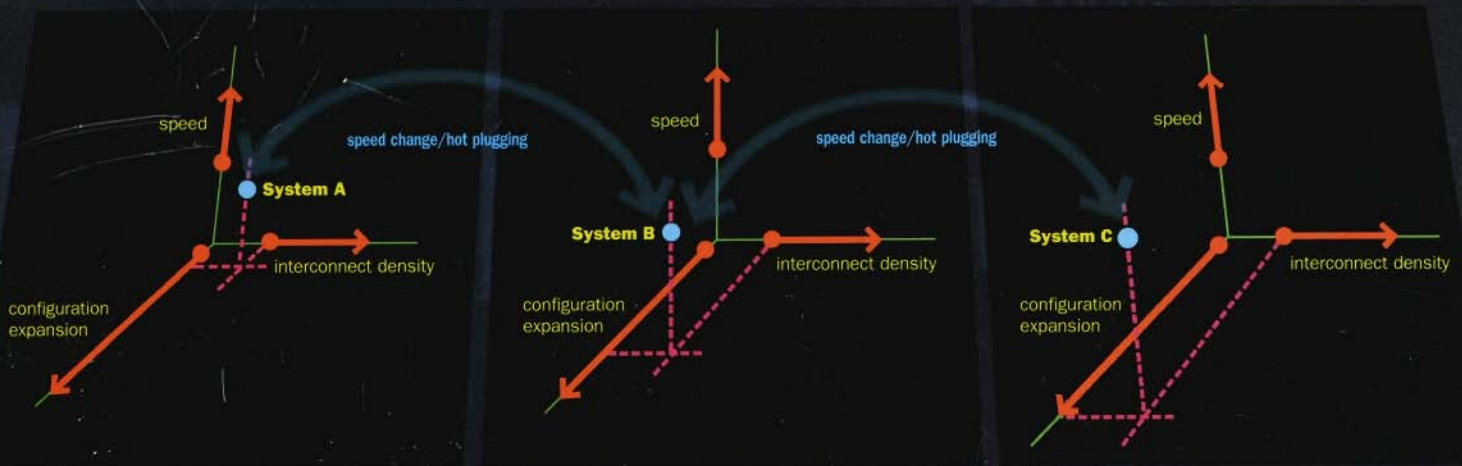
configuration expansion

speed change/hot plugging

configuration expansion

speed change/hot plugging

configuration expansion



UltraSCSI

UltraSCSI

UltraSCSI

UltraSCSI

UltraSCSI

UltraSCSI

UltraSCSI

UltraSCSI

Volume 9 Number 3
1997

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

Circulation
Catherine M. Phillips, Manager
Kristine M. Lowe, Administrator

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

Advisory Board
Samuel H. Fuller, Chairman
Scott E. Cutler
Thomas F. Gannon
Donald Z. Harbert
William A. Laing
Richard F. Lary
Alan G. Nemeth
Robert M. Supnik

Cover Design

Recent advances in physical technology have significantly improved the capabilities of parallel SCSI in three parameters: speed, interconnect density, and configuration expansion (device count, length, topology, control). On the cover, the graphs represent three systems with different sets of parameter values, that is, three unique points in the speed-size-configuration space. Hot plugging of devices and bus segments and dynamic speed changes can decrease or increase the parameter values without system disruption. The opening paper in this issue describes these advances in parallel SCSI technology.

The cover design is by Lucinda O'Neill of the DIGITAL Industrial and Graphic Design Group. The editors thank author Bill Ham for his help in developing the cover concept.

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.**

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 978-264-7556.

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.

Copyright © 1998 Digital Equipment Corporation. Copying without fee is permitted provided that such copies are made for use in educational institutions by faculty members and are not distributed for commercial advantage. Abstracting with credit of Digital Equipment Corporation's authorship is permitted.

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

ISSN 0898-901X

Documentation Number EC-P8826-20

Book production was done by Quantic Communications, Inc.

The following are trademarks of Digital Equipment Corporation: AlphaServer, AlphaStation, CI, DECnet, DECNIS, DIGITAL, the DIGITAL logo, DIGITAL UNIX, and PowerStorm.

Betamax is a trademark of Sony Corporation.

CRAY is a registered trademark of Cray Research, Inc.

Direct3D is a trademark of 3Dlabs, Inc. Ltd.

Microsoft, Visual C++, Windows, Windows 95, and Windows NT are registered trademarks and NetMeeting is a trademark of Microsoft Corporation.

Netscape Communicator is a trademark of Netscape Communications Corporation.

Pro/ENGINEER is a registered trademark of Parametric Technology Corporation.

TotalView is a trademark of Dolphin Interconnect Systems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.

WinFrame is a registered trademark of Citrix Systems, Inc.

Contents

Foreword	Richard Lary	3
Recent Advances in Basic Physical Technology for Parallel SCSI: UltraSCSI, Expanders, Interconnect, and Hot Plugging	William E. Ham	6
Development of Router Clusters to Provide Fast Failover in IP Networks	Peter L. Higginson and Michael C. Shand	32
Shared Desktop: A Collaborative Tool for Sharing 3-D Applications among Different Window Systems	Lawrence G. Palmer and Ricky S. Palmer	42
Challenges in Designing an HPF Debugger	David C. P. LaFrance-Linden	50

Editor's Introduction

This issue of the *Digital Technical Journal* presents papers on a range of computing subjects, beginning with recent advances in storage technologies, followed by network router cluster enhancements, new desktop software for sharing 3-D applications across platforms, and an experimental High Performance Fortran debugger.

DIGITAL's storage engineers have been leaders in the definition of the parallel small computer system interface (SCSI) ANSI standards and in related technology improvements. Bill Ham's paper focuses on four advances in the physical features of SCSI that resulted in major increases in SCSI capabilities and minor disturbances when incorporated in existing installations. The discussion spans developments from SCSI-2 through UltraSCSI, including speed increases in the synchronous data phase; longer, more complex configurations enabled by bus expanders; physical versatility inherent in a decreased size of the interconnect; and dynamic removal and replacement of devices on an active bus (hot plugging).

The subject of our next paper is networks, and the emphasis of the engineering is on customer requirements for reliability and availability. Router clusters, described here by Peter Higginson and Mike Shand, were developed to provide fast fail-over response in IP networks and are defined as a group of routers on the same local area network (LAN) providing mutual backup. New router cluster protocols and mechanisms restrict the loss of service that results from a failure on the network, specifically on networks requiring high availability, such as telecommunication and stock exchange networks. The authors analyze failure cases and present the solutions that reduced service-loss time from approximately 30 to 45 seconds to 5 seconds in both LAN and WAN environments.

Collaboration software for desktop systems can be broadly defined to encompass a range of capabilities, from a simple transfer of data between users, such as e-mail sent over a network, to real-time sharing of text, graphics, and audio and video data. Larry and Ricky Palmer have designed a software product, called Shared Desktop, for users who want to share three-dimensional graphics applications and audio across networks. Notably, the design differentiates itself by supporting multiple operating systems, currently enabling real-time interoperation among Windows and UNIX systems. The authors discuss the decision to create a "viewport," which is a part of the desktop screen, and issues they addressed during implementation, including protocol splitting, screen capture and data handling, and dissimilar frame buffers. They conclude with ideas for possible enhancement of the product in the future.

In a previous issue of the *Journal* featuring technical computing topics (vol. 7 no. 3), Jonathan Harris et al. described DIGITAL's Fortran 90 compiler that implements High Performance Fortran version 1.1, a language for writing parallel programs. An outgrowth of that work is an experimental debugger, code-named Aardvark, that "reconstructs" for the HPF programmer a single source-level view, even though the program has several flows of control and the data are distributed. David LaFrance-Linden discusses the challenges faced in creating the debugger and describes useful techniques and concepts, such as logical entities, that can be generally applied to debugger design.

Readers interested in past issues of the *Journal* are invited to visit the *Journal* Web site at http://www.digital.com/info/dtj/. Our next issue will address such topics as optimization of NT executables on Alpha, a new graphics program, and VLM. A Special Issue on programming languages and tools is being developed for publication in the fall of 1998.

Jane C. Blake
Managing Editor

Foreword

Richard Lary
DIGITAL Storage Technical Director

Welcome to the winter 1997-98 issue of the *Digital Technical Journal*. This issue does not have a single theme; it contains a potpourri of papers on a wide range of technical topics. This provides the foreword writer with a small gift and a not-so-small headache.

The gift is the opportunity to tout the continuing fecundity of DIGITAL's engineering community. All the papers in this issue of the *Journal* come from product development groups in DIGITAL, and all the technology described herein is directly applicable to the problems of using computers in the real world. The papers themselves cover a wide range of topics: designing storage buses and their infrastructure; building IP routers that reduce network delays caused by link or router failure; sharing 3-D graphical and audio data across networks of computers with different windowing systems; and debugging programs written in languages that incorporate data parallelism.

The headache, of course, stems from this very diversity. Any attempt to derive some set of common underlying principles other than "make better stuff" from this collection is doomed to sophistry. And my technical background is too narrow to provide any significant embellishment to any of the papers outside the domain of storage systems. So, with apologies to the other authors, I am forced to restrict my comments to what I know —the background and impact of Bill Ham's work on advances in parallel SCSI which are presented in his paper in this *Journal*.

Bill Ham's paper not only describes a significant technical achievement; it illustrates DIGITAL's shift from engineering proprietary storage systems to engineering open storage systems.

The SCSI bus was developed during the early 1980s as one of many attempts to standardize the interface to storage devices. It succeeded beyond the expectations of its developers, largely because it supported a device model that was abstract enough to be extensible but inexpensive enough to be implemented in the technology of the time. For all its advantages, however, SCSI suffered from poor engineering at the physical level. This was a direct result of the way it was developed. The diverse corporate representatives that defined SCSI did not have the time or money to specify and build custom bus infrastructure components (transceivers, cables, terminators, etc.), so they used commonly available parts. A lack of sophistication in specifying physical interface parameters resulted in a specification that allowed too much component variation. As a result, it was difficult to build reliable, multi-box systems using SCSI.

DIGITAL's attitude towards SCSI during this period was to ignore it and hope it would go away. We had designed our own proprietary Digital Storage Architecture (DSA), which utilized an abstract and extensible device model and also incorporated many large system features, including a robust physical interconnect. We controlled the design and manufacture of all DSA components and could thus guarantee that they all met tight architectural specifications. Moreover, DSA was a key enabling technology for VMS Clusters, the individual DSA components were competitive with their counterparts

from the proprietary storage architectures of other large systems companies, our customers were happy, and the storage business was profitable. We were feeling quite pleased with ourselves—and we were profoundly ignorant of the power of a successful open market standard, since one had never existed in the storage world.

During the latter half of the 1980s, SCSI grew steadily in popularity until it dominated the workstation and small-server markets. These systems had at most a few disk drives on them, and SCSI's signal integrity problems were manageable in that context. They were not manageable in the larger and more demanding data center systems, and so SCSI was not used there. The SCSI standards group was aware of the bus's deficiencies, however, and as the decade progressed, the group made amendments to the standard to eliminate many of them. By the turn of the decade, several independent subsystem vendors were selling subsystems utilizing SCSI devices as storage for large DIGITAL systems. These subsystems did not, in general, have the features, performance, or robustness of our subsystems, but they were significantly cheaper and improving all the time. By 1991, it had become obvious to us that we would not be able to compete with these systems in the long run. They were leveraging an entire industry's investment and talent and were reaping the cost benefits of high-volume manufacturing; whereas we had to design and manufacture (at relatively low volume) every component of every DSA system ourselves.

Our position was untenable. We had to change our strategy and embrace the bus that we had so studiously ignored.

We designed a modular packaging architecture for SCSI devices (known commercially as StorageWorks) and a set of storage array controllers that interfaced these devices to our systems (and systems from other major vendors as well). We also became active participants in the SCSI standards process. Where DIGITAL had previously sent one or two engineers to SCSI standards meetings strictly to gather information, we started to send up to half-a-dozen engineers to listen, learn, participate in debate, help with the grunt work of the standards process, and make proposals to amend or extend the standard in directions useful to us and our customers.

Our new modular packaging design allowed our customers to install and remove storage devices themselves and to migrate storage devices between systems, even between systems built by different system vendors. This modularity proved to be a very valuable feature to our customers. However, it required us to build a physical infrastructure for the SCSI bus that had the robustness needed by our large systems and that could accommodate a great deal of variability in configuration, and to use a bus that was known to have residual signal integrity problems in its physical interconnect. We were understandably worried about this, worried enough to charter a small group of engineers as a SCSI Bus Technical Office (SBTO) within the storage group, and to develop short-term configuration guidelines for our packaging architecture and long-term technical proposals for the SCSI physical bus architecture. Bill Ham has been the head of SBTO since its inception and has also been our representative to the SCSI committee on all matters relating to the physical bus interconnect.

In the summer of 1993, Bill completed a study of the signal integrity issues surrounding parallel SCSI. His conclusions were startling. The SCSI standards committee had, over the years, made enough improvements in the basic transmission line characteristics of the SCSI bus that most of the remaining signal integrity problems were due to the variations in component parameters allowed by the SCSI specification. Exercising tighter control over component variation—through building selected components or through purchase specifications with our suppliers—would not only produce excellent signal integrity in our packaging but would allow the maximum clock rate of the bus to be doubled while maintaining excellent signal integrity and backwards compatibility with existing SCSI devices. Bill's results also indicated that the maximum clock rate could be increased even further, with more work.

This discovery came at a critical time in the evolution of the SCSI standard. Much of the SCSI standard committee's effort in the early part of the 1990s was being spent in modifying the SCSI standard so that serial buses could carry the higher level SCSI bus protocols. The committee had started this work under the

assumption that parallel SCSI was "out of gas" in performance, and the new serial bus variants would supplant it by mid-decade. However, by 1993 not only was the definition and implementation of the serial bus going slower than expected, but there were three independent and incompatible serial bus proposals, each with unique useful features and unique drawbacks, each with a cadre of supporters among the industry representatives. The market would ultimately choose which serial buses would thrive; but it was highly unlikely that all three would thrive. Storage vendors that made the wrong bus choice would suffer for it. Most galling to the technophiles among us, the market's choice could not be predicted from the technical merits of the contenders. If it could, we'd all have Betamax VCRs in our homes today.

So, DIGITAL decided to have Bill present his results to the SCSI committee at its November 1993 meeting and recommend that the committee extend the SCSI specification to allow the bus to run at up to twice its old maximum clock rate if the components in the physical interconnect met the tighter specifications. Our motive in doing this was purely selfish: we were not ready to choose among the serial bus proposals, yet we would soon need more performance than parallel SCSI could offer. A higher performance parallel SCSI would allow us to improve our storage subsystem performance without having to stake our fortunes on a potential Betamax.

Bill's presentation at the SCSI committee meeting was met with enthusiastic approval. It turned out—surprise!—that other system vendors were feeling as uneasy as we were about the serial SCSI buses. The proposal, christened UltraSCSI, was adopted as an extension to the parallel SCSI standard. Bill Ham and the SBTO then worked with component vendors and the SCSI committee to develop the thinner cables, smaller connectors, and SCSI expander circuits described in his paper, all with the aim of keeping parallel SCSI as a desirable alternative to the serial SCSI buses. Today, four years after its committee debut, UltraSCSI is solidly entrenched in the storage market. In fact, storage market analysts are now projecting that the combined volume of devices on all serial SCSI buses (yes, there are still three, but the market has already picked one, Fibre Channel, as the winner) will not exceed parallel SCSI device volumes until early in the next century. And the SCSI committee has finished extending the parallel SCSI specification to achieve a second doubling of maximum bus clock and is in the midst of defining a third doubling.

Without hyperbole it can be said that the technology embodied in Bill Ham's paper has directly affected the course of the computer storage industry, and it continues to affect positively DIGITAL's position in that industry. Enjoy reading the paper and those that follow it in this issue.

William E. Ham

Recent Advances in Basic Physical Technology for Parallel SCSI: UltraSCSI, Expanders, Interconnect, and Hot Plugging

DIGITAL uses SCSI technology in most of its storage products and consequently has led major standards and industry bodies to improve the technology in the following areas: increased synchronous data phase speed beyond fast SCSI; longer, more complex electrical configurations by means of expander circuits; versatile and more manageable connectivity through a smaller, improved physical interconnect; and dynamic device insertion and removal. Data phase transmission rate extension is achieved through understanding and controlling silicon chip timing and transmission media parameters. Using expander devices to confine transmission line effects to shorter segments allows large increases in the maximum distance between devices and in the device population within the same SCSI domain. Expanders enable complex, hublike configurations to be created without changing existing SCSI devices or software. The use of 0.8-millimeter connector technology and consideration of cable losses has reduced the physical size of the external shielded interconnect by approximately two thirds, decreased the number of parts required to support complex configurations by a factor of 10, and increased the interconnect density to the same level used in serial SCSI. Finally, the mating and demating events that occur during device insertion and removal produce a spectrum of small, undetectable, electrical disturbances on the active bus that appear to be limited by the physics of the media and device capacitance.

Introduction

Parallel Small Computer System Interface (SCSI) is the workhorse technology for most of the storage applications in DIGITAL products today. This device and interconnect technology spans all system offerings from the simplest to the most complex. SCSI was introduced to the higher-end products in the early 1990s as the open systems follow-on to the DIGITAL proprietary Digital Storage System Interconnect (DSSI) and Computer Interconnect (CI) technologies.

As system demands have increased, SCSI has evolved to meet the needs. DIGITAL has made considerable contributions to the technology and led the effort to achieve industry standardization. This paper details the most significant developments in the physical features of parallel SCSI technology over the last several years that have allowed it to continue to serve DIGITAL customers in an effective, competitive way. The discussion targets the following four important areas:

1. Speed increases in the synchronous data phase, which resulted in the ANSI definition of UltraSCSI (Fast-20 SCSI) technology[1]
2. Development of software-invisible circuits, generally called expanders, that enable segmentation of SCSI domains into easily managed pieces
3. New connector and cable technology, namely the Very High Density Cabled Interconnect (VHDCI) device, that decreases the interconnect size and complexity by many fold[2]
4. Dynamic removal and replacement of devices on an active bus, which is referred to as hot plugging

DIGITAL made substantial contributions in the four areas. This work included creating the expander and interconnect standards projects; leading the working groups that defined the Fast-20, expander, and interconnect standards; providing data for the Fast-20 and hot-plugging projects; and proposing and gaining approval for the hot-plugging standard.

The author has taken a phenomenological approach throughout, because in most cases there are too many unknowns to achieve a rigorous analytical result. This

paper focuses on developments from SCSI-2 through UltraSCSI and specifically does not address the new Low Voltage Differential (LVD) technology being introduced for the highest-speed applications.

Pedigree

SCSI is defined in several ANSI standards[1,3,4] and in the material that was developed to create these standards.[5,6] The standards were generated over the last decade through a cooperative effort of approximately 60 major companies in the computer and computer support industry. As a result of this pedigree, the prime directive for SCSI technology is interoperability of devices designed and manufactured by different companies.

The details of the physical designs used to implement SCSI may not be visible to users and researchers; these details contain much of the marketing and technical differentiation between the products of the participating companies and are therefore hidden in the silicon design. The behavior at the device connector pervades the SCSI specifications. The basic assumption is that as long as the properties are compatible at these connectors, device substitution is possible. Thus, SCSI devices may be both interoperable and of different designs.

Basic Architecture

This section reviews the basic architecture of parallel SCSI. The SCSI bus is a parallel, multidrop, wired-OR configuration.

Signal Multiplexing and Phases The parallel signal construction of the bus allows multiplexing of some signals during different phases of communication so that the same signal lines may have very different functions in different phases. The physical behavior of signals is usually limited by the phase during which the shortest pulses are used and the demands for signal integrity are the highest. The limiting SCSI phase is the data phase (payload phase) that is executed with the highest synchronous rate. For UltraSCSI, this peak repetition rate is 20 megahertz (MHz). Table 1 contains the generally accepted terminology related to data phase speeds.

Because of the wired-OR property, each signal in the bus must be driven to a known state even if no SCSI device is actually driving the signal. SCSI uses the logical 0 state (negated state) as the undriven state and uses the bus terminators to drive the signal to this state in the absence of any driving devices. The device signal drivers must overcome this terminator-driven logic state of 0 in order to send a logical 1 (asserted state) onto the signal line.

SCSI signals must support all frequencies, from statically driven by the terminators only (DC) to the third harmonic of the fastest signal edge in the synchronous data phase. In many cases, the same wire must support all these frequencies at different times during the SCSI protocol.

The highest signal edge slew rates for UltraSCSI are approximately 500 millivolts per nanosecond (mV/ns). A 2-volt (V) transition requires approximately 4 ns/5.4 ns/meter (m) = 0.74 m for a signal edge (assuming 5.4 ns/m as the propagation velocity of the signal edge). Therefore, some relief exists because the connectors and cable assembly terminations are much smaller than the signal edge length; the connectors and terminations do not need to have carefully controlled characteristic impedance properties. This allows the use of the technology available in the connector and cable assembly industry to optimize the interconnect properties without the considerable design, manufacturing, and test burden imposed by controlled impedance requirements.

Transmission Modes The transmission mode of a SCSI bus is determined by the properties of the terminators that, by definition, constitute the ends of the bus. Terminators also supply most of the energy required to operate the single-ended transmission-mode devices and additionally provide the required matching

Table 1
Terminology for Data Phase Speeds

Data Phase Speed Name	Maximum Transfer Rate (Million transfers/second)[1]	Maximum Byte Rate (Narrow) (Megabytes/second)	Maximum Byte Rate (Wide) (Megabytes/second)
Asynchronous	Unspecified	Typically ~ 3	Typically ~ 6
Slow (synchronous)	5	5	10
Fast (synchronous)	10	10	20
Ultra (synchronous)[2]	20	20	40
Ultra2 (synchronous)[3]	40	40	80
Ultra3 (synchronous)[4]	80 to 100	80 to 100	160 to 200

[1]One transfer is 1 byte in narrow mode and 2 bytes in wide mode; 1 byte equals 8 data bits plus 1 parity bit.
[2]Ultra is synonymous with Ultra1 and Fast-20.
[3]Ultra2 is synonymous with Fast-40.
[4]Rates not yet finalized; Ultra3 is synonymous with Fast-80 or Fast-100.

to the characteristic impedance of the transmission line. In differential SCSI, the terminators provide a small portion of the overall energy required to operate the bus; the differential drivers supply the remainder of the energy.

Drivers that want to transmit an asserted state must overcome the biasing provided by the terminators. The drivers operate locally on the bus and alter the state in their immediate vicinity when they switch on and off. For single-ended SCSI, the 0 state is approximately 2.5 V and the 1 state is approximately 0.5 V. For high-voltage differential SCSI, the 0 state is approximately −1 V to −2 V, and the 1 state is approximately 2 V. (The difference between a state 1 and a state 0 is higher with differential—typically, approximately 4 V.)

For single-ended transmissions, the drivers operate on energy previously stored in the bus by the terminators. This energy is mostly electrostatic energy in the charge stored in the capacitance of the transmission line for negated states and electromagnetic energy in the current flowing through the inductance of the transmission line for asserted states. Ultimately, the terminators will set the state back to negated after the drivers cease to source or sink current; however, this only happens after the round-trip propagation delay from the driver to the farthest terminator if the bus does not have matched characteristic impedance properties.

Approximately the same energy transformations occur for differential SCSI, but significant current is supplied by the drivers for both the asserted and the negated states.

Multidrop Requirements The multidrop architecture requires a continuous low-resistance path called the bus path between the terminators and allows devices to be attached to this path. The number and properties of these attached devices vary widely because of many factors including the speed of operation, the overall length of the bus, and the transmission mode. Attached devices always disturb the transmission line properties of the bus path; the key to successful operation is in the management of the magnitude of these disturbances.

Generally, the more capacitance or electrical length the device has, the more disruptive it is. Placing devices too close together along the bus path can cause them to appear electrically as a single super disruptive device. Placing them too far apart can result in an overall bus length that is too long.

Wired-OR Glitches During the arbitration phase, when the SCSI devices decide which devices will be sending payload data to or from each other, multiple devices may assert the same control line (BSY) at the same time. Each device that wishes to communicate asserts both the BSY line and its respective device identification (ID) line. After examining the asserted ID lines to determine which device has the highest ID, all but the device with the highest ID release the BSY line. This leaves only one device, the winner, asserting the BSY line. While the current in the BSY line is readjusting itself from a multiple-driver asserted condition to a single-driver asserted condition, noise pulses (called wired-OR glitches) propagate throughout the length of the signal line and may be detected collectively as an erroneous phase. Therefore, one of the architectural limits for parallel SCSI is the time required for these wired-OR glitches to settle. This bus settle time is set by protocol at 400 ns and must be interpreted as a round-trip propagation time when using a simple SCSI bus. Allowing some time for propagation through driver and receiver chips yields a maximum physical length for a simple bus of 25 meters.

Areas of Improvement

Thus, the opportunities for improving SCSI derive from appropriately managing the transmission lines, taking advantage of the multidrop architecture offered by a parallel wired-OR structure, using state-of-the-art technology from the interconnect and silicon industry, and making innovative use of the time required for the wired-OR glitches to settle. These techniques are the basis of the development by DIGITAL in the four areas addressed in this paper.

Speed increases in the synchronous data phase are based primarily on increasing the timing precision in the silicon transceivers by using newer silicon technology. The interconnect properties remain largely unchanged from those used for fast SCSI.

Circuits that enable segmentation of SCSI domains into easily managed pieces are based on systematic isolation of transmission line properties and use of wired-OR noise pulse properties. No software, interconnect, or device changes needed to use these circuits.

New connector and cable technology is based on an innovative 0.8-millimeter (mm) ribbon-style connector technology that optimizes the total SCSI electrical requirements with the capabilities of cable and connector design.

Dynamic removal and replacement of devices on an active bus, i.e., hot plugging, is based on the multidrop architecture, which enables devices to be added or replaced without affecting continuity between other devices. Hot plugging depends on understanding and managing the electrical disturbances created during the insertion or removal.

The remainder of this paper provides details of these four areas of improvement. The end result of these extensions to the basic physical architecture of parallel SCSI is a major increase in its capabilities, accompanied by only a very minor disturbance to the installed base, especially the software.

Increasing the Synchronous Data Phase Speed

Beginning with the SCSI-2 standard, the synchronous transmission mode is available for transferring payload data between SCSI devices. The devices select this mode by mutual agreement before any synchronous data is passed. The agreement is achieved by using the asynchronous transmission mode, which is slow but usually reliable.

The synchronous data phase uses the DATA and PARITY bit lines for the data and either the REQ or the ACK control line as a signal that the receiver uses for capturing the data. The term synchronous derives from a specified timing relationship between the bit line signal edges and the REQ or ACK signal edges. (The falling edge of the ACK signal is used when the data phase transmission originates from the SCSI initiator, and the falling edge of the REQ signal is used when the transmission originates from the target.) There is no synchronous relationship between the internal timing references on different SCSI devices, so the receiver must buffer the received data before introducing the data into its internal data management structure. This buffering is usually accomplished by means of a first in first out (FIFO) circuit that uses the REQ or the ACK signal as the latching signal for the incoming data. For convenience, in this paper we only refer to the ACK signal, with the understanding that the same discussion applies to the REQ signal when it is used as the data-latching signal.

Since only the falling edge of the ACK signal is used in the presently specified SCSI versions and an ACK signal is required for every data transfer, it follows that the ACK signal cycles at least twice as fast as the data bits. When a continuous stream of transfers is transmitted, the ACK signal is a regularly repeating signal, nominally, a square wave. An alternating 1/0 pattern produces the highest fundamental frequency for the data bits at half the frequency of the ACK signal. Therefore, the ACK signal requires careful attention since it is the most demanding on the transmission process.

The focus of this section is to examine how the speed of the synchronous data phase was increased by a factor of two to achieve the Fast-20 (UltraSCSI) specification.

Status before UltraSCSI

In 1993, the SCSI-2 standard[3] had been in place for two years, and a follow-on standard called SCSI-3 Parallel Interface (SPI)[4] was technically stable. SPI had been created largely because the specifications in the SCSI-2 standard were not effective in implementing the single-ended version of the synchronous transmission (10 megatransfers per second). The differential version specified in SCSI-2 worked well but was much more expensive in cost, power, and space than the single-ended version. Therefore, most of the interest was in making the fast single-ended version work adequately.

Taking single-ended SCSI from asynchronous and slow synchronous (5 megatransfers per second) to the fast synchronous technology was difficult. The prevailing opinion was that the SPI standard represented the final improvement to parallel SCSI. This view set the stage for a number of alternate physical technologies based on the serial point-to-point transmission schemes used in communications technologies, e.g., Fiber Distributed Data Interface (FDDI) and Ethernet, to be used for higher-performance storage applications.

DIGITAL's Storage Bus Technical Office had seen many instances of difficult implementations that were the result of less-than-optimal understanding and management of the specification margins. No credible study had been presented on the margins available in SCSI, so the thrust was to create baseline characteristics of multidrop parallel SCSI to determine where unused margin might exist.

Little data was available on the precise reasons why specific implementations of fast synchronous SCSI did not work. The system would hang or report various error messages with almost no indication of the basic causes. A method that could report margin to failure and mechanism of failure was needed to unravel this situation. Therefore, the approach DIGITAL took was to step back from full SCSI implementations and to examine the pieces without the encumbrance of the SCSI protocol.

One of the most mysterious areas was the behavior of SCSI receivers. The SCSI-2 and SPI specifications used bipolar transistor-transistor logic (TTL) levels as the basic receiver input levels. Almost all SCSI devices were being designed with complementary metal-oxide semiconductor (CMOS) technology, so the differences between the receiver properties presented a key opportunity for hidden margin. Other unknown areas were jitter, cross talk, skew, ground offset, effects of stubs, and worst-case configurations.

DIGITAL built a special test environment to systematically examine each piece of parallel SCSI. The environment was named the PBDIT, an acronym for parallel bus data integrity tester. This test environment made it possible to systematically examine the real margins to failure for the key pieces and to develop the confidence that SCSI could be used at elevated speeds and be made highly robust at the slower speeds.

Special Test Environment

The test environment was built to allow known data patterns to be transmitted across a SCSI device, into SCSI transmission media, and then into another SCSI device. The same data pattern is loaded into both sides so the receiver knows exactly what data it is supposed

to receive. The transmitting side is called the exciter, and the receiving side is called the comparator. Received data is committed to the comparator by using one bit line as the latching ACK signal in a manner exactly like that specified in synchronous SCSI transmissions. The test environment allows the position of the ACK signal to be adjusted with respect to the data signal edges.

Since the comparator knows the data pattern that is transmitted, it is possible to isolate the precise data bit that caused the transmission error. This kind of error-directed methodology has found widespread use in the integrated circuit industry.

Other features of this test environment include detachable load boards that contain the SCSI drivers, terminators, receivers, connectors, or any other physical media-dependent components. The minimum requirements for a load board are that the exciter contain the SCSI driver and a connector and that the comparator contain the SCSI receiver and a connector. Other components may be placed between the load boards for different test conditions. The SCSI driver must have accessible points for the exciter logic, and similarly, the SCSI receiver must have output points to drive the comparator. These requirements eliminate drivers and receivers that are imbedded within chips with other functions. Fortunately, separate SCSI drivers are available for both single-ended and differential versions. (The differential versions normally use separate chips, but only a few choices are presently available for the separate single-ended versions.)

The test environment is useful for developing the understanding of operating mechanisms and for measuring the margins for specific hardware configurations. This environment is not useful for deriving specifications, since the performance at the specified interfaces, i.e., the device connectors, is not directly observable. Oscilloscope measurements provide the basis for setting compliance specifications, since these measurements can be performed at the connectors. The basic question that needed an answer was, Can parallel SCSI be operated at elevated speeds with reasonable margin to failure? DIGITAL optimized the special test environment to answer this question. Other specifications that would be necessary to ensure interoperable operation between UltraSCSI devices could be derived if it appeared possible to achieve the end result.

The data pattern loading and digital control of the exciter and the comparator were achieved through optically coupled means. This allowed the ground offset voltage to be adjusted between the driver and the receiver without compromising the operation of the logic.

The data flows only from the exciter to the comparator. If bidirectional information is desired, the physical connections between the exciter and comparator have to be reversed. This scheme leaves untested the cross-talk effects on the REQ signal that is traveling in the opposite direction to the ACK signal (if ACK is synchronized with the data as in a write operation). Separate measurements are necessary to examine this issue. Cross talk into other control lines is addressed by holding these lines constant in the data pattern transmitted.

The SCSI standard deals with the REQ cross-talk issue by requiring that the data lines be physically separated from the REQ and ACK lines in the transmission media. Measurements not reported in this paper have confirmed negligible speed-related cross talk into the REQ line.

Up to 27 pairs of 3-byte-wide lines (wide SCSI uses only 18 pairs for high-speed transmissions) can be tested with the special test environment. Figure 1 is a functional diagram of the test environment. The SCSI terminators are shown as separate from the load

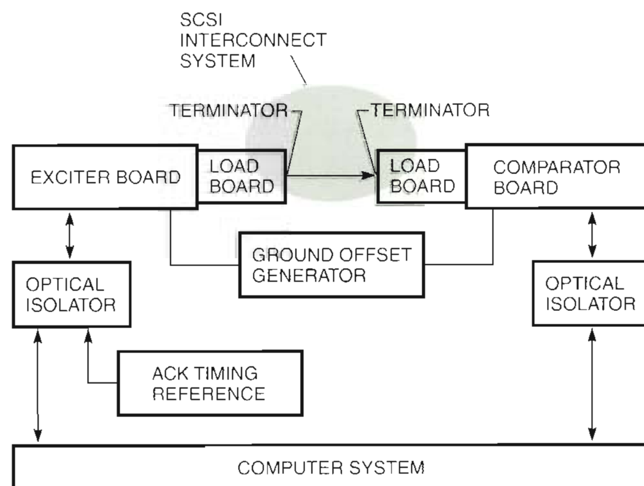


Figure 1
Special Test Environment

boards in this case. A key feature of this kind of testing is that the test does not necessarily stop when an error is detected. In fact, the environment may detect errors 100 percent of the time. This acceptable behavior allows mapping of the complete bit-error response of the system.

Sample Data from the Special Test Environment The test environment allows a multitude of tests to be performed. The test scheme described in this section is the one that was used to establish the basic timing margins available from normal SCSI silicon, cables, connectors, and terminators.

A random repeating data pattern with 16 thousand different bit combinations was used as the basic data pattern. This pattern was transmitted over a period of time, and the number of errors detected was recorded. In this test, an error is defined as one or more bits in the received data transfer that do not match the transmitted bit. To acquire a new error rate data point, the transmission test is repeated by using exactly the same number of transfers in the same time period with the same data pattern but with some test parameter changed.

Virtually any parameter can be varied for different tests. For a given physical configuration, the most useful parameter for determining the timing margin is the position of the ACK pulse with respect to the data edges. The basic data then becomes the number of errors detected and the position of the ACK pulse edge.

There are two basic random variables operating in this scheme: the data pattern and the jitter induced by non-data-dependent sources. It is easy to separate these two variables by using extremes in the data pattern: very few transitions and the maximum number of transitions (every data edge has a transition, i.e., alternating 1/0 pattern). Although this level of precision is available, we will see that we really do not need to bother for parallel SCSI at the maximum UltraSCSI rate.

Figure 2 shows a typical error rate plot from a simple single-ended configuration made from ordinary SCSI interconnect hardware and transceivers being tested at the maximum UltraSCSI rate. Each data point represents a 3-second sample (60 million transfers) at each ACK position. The ACK position is incremented in 0.1-ns steps for a total of 240 independent tests in the plot. To minimize the testing time, we tested only the time ranges from −3 to 9 ns and 44 to 56 ns. The individual data points are not distinguishable in this presentation, and there is very little scatter between neighboring points. In Figure 2, the error rate of 1 is used to indicate that no errors were detected, since the log of 0 is not easy to plot.

Examination of the raw data reveals that the plot is monotonic in detected error rate to the fourth decimal place. This indicates an extremely predictable situation as far as behavior of the same set of hardware is concerned. That is, there is virtually no Gaussian jitter present, and a SCSI system could be designed to be quite reliable and stable at the maximum UltraSCSI rate.

Extending the sample period to 5 minutes made no difference in the position of the key features. Using the 3-second sampling time, the entire data set could be acquired automatically in approximately 12 minutes.

The onset of errors is extremely sharp as the ACK position approaches the critical position. One hundred picoseconds changes the observation from 0 to 864 errors near the 8-ns position. On the other end, the 50.1-ns time produced 7 errors, and the 50.2-ns time produced 425 errors. No errors were detected at any of the times between 50.1 ns and 7.9 ns. This data shows that there are no strange effects that prevent SCSI from operating at the maximum UltraSCSI rate.

As the ACK position proceeds into the region of more errors, a condition is finally reached in which *all* the transfers have errors. On the one hand, the probability that one transfer has the same data content as its

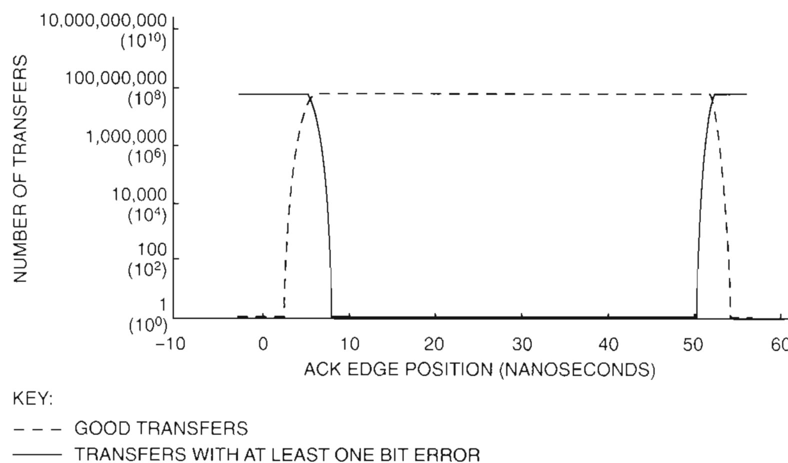


Figure 2
Typical UltraSCSI Error Rate Plot

neighbor's is very small with this random data pattern. On the other hand, since a random data pattern is being used, there is a reasonable chance that a bit will actually match that transmitted in one state but not in the other state. The random data pattern tends to spread out the time between the first error and the last good transfer. In the limit, for perfectly random data, this time is a measure of the total timing imprecision in the system.

This imprecision includes skew in the exciter and comparator boards, in the SCSI drivers and receivers, and in the cable transmission media (including loads, if any), and all forms of jitter. For the test conditions shown in Figure 2, the total difference is 3.6 ns near the 5-ns point and 5.4 ns near the 52-ns point. This shows that the skew specifications in the SCSI standard are overspecified as compared to actual hardware performance.

The data shown in Figure 2 is representative of a large variety of configurations up to approximately 3 meters long and loaded or up to much longer point-to-point lengths (20 meters or more [see Figure 6]). The error-free window can be made to collapse by adding too many loads or by using the wrong impedance cable, improper terminators, receivers with the wrong threshold voltages, or other bus component and configuration parameters. However, the details of the actual hardware and configuration do not affect the basic conclusion derived from Figure 2, namely, that a great deal of timing margin is available at the maximum UltraSCSI rate when ordinary SCSI hardware is used.

To put this into perspective, basic gigabit-per-second serial transmissions with approximately twice the basic bandwidth of UltraSCSI have bit times of about 1 ns and timing margins of a few hundred picoseconds. UltraSCSI has an effective margin window of a few tens of nanoseconds. This represents two orders of magnitude more margin for the parallel SCSI application.

The initial errors usually originate from the same bit. This bit is the one with the most unfavorable timing skew with respect to the ACK signal. The cliff is not perfectly sharp because there is a 50 percent chance that the data transmitted is the same as that expected even under the error case and, more importantly, because there is some level of jitter present. It is this jitter that softens the cliff. Thus, the first errors detected happen when the skew of the weakest bit adds to the tail of the jitter distribution. Only a few errors are present because only a small part of the jitter population extends far enough to trigger the error. SCSI systems will experience virtually no errors because of these mechanisms in service if one operates 1 ns or more away from an error cliff.

Note that these results from the special test environment almost always yield margins higher than those calculated from a set of interoperability specifications. This is because the interoperability specifications must allow margin for each piece, and the special test environment reports the integrated result from many pieces in the complete SCSI connection.

Higher Speeds The main effect of further increasing the transfer rate above the maximum UltraSCSI rate in the same set of hardware is to change the time position of the onset of nonzero error rates and to narrow the error-free region. Figure 3 shows an example of data from Fast-40 transmissions using separate high-voltage differential transceivers on each bit. (This data was acquired by DIGITAL's Storage Bus Technical Office in 1994.)

The error-free zone has narrowed to approximately 15 ns, and the time between first error and 100 percent errors has widened on both sides, but still no uncontrolled regions exist. This strongly suggests that at least Fast-40 transfer is possible with no major technology changes in the interconnect.

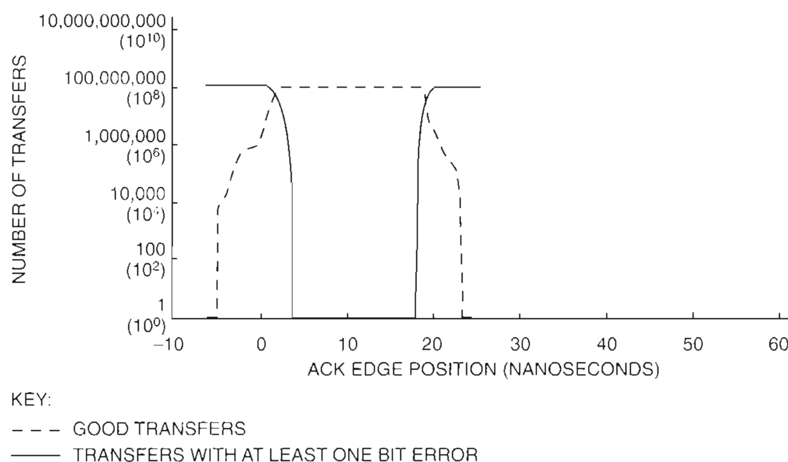


Figure 3
Fast-40 Error Rate Plot

Additional Tests Other tests that are useful with the special test environment are ground offset effects, terminator power effects, correlation of time domain plots on the signals with error rate distributions, hot-plugging testing (which results in good error detection), and comparison of the impact of different cables and transceivers. Test results of this nature are not included in this paper because the impact of these variations depends on many parameters and the results may not be generally applicable.

Timing Specification Methodology

With the increased emphasis on timing precision for UltraSCSI technology, it was necessary to introduce better specifications for the measurement of timing parameters than those in the SCSI-2 and SPI standards. Figure 4 shows the precise measurement points and features used for the specification of single-ended UltraSCSI signals.

The effects of the finite slew rate on the signal edges are accounted for largely by specifying the voltage levels that coincide with the receiver input levels. Thus, the setup time ends when the receiver is able to detect an asserted state at 1.3 V, and the asserted period begins when the asserted state has been detected. On the negation side, the signal must rise to at least 1.6 V before the receiver can detect a negated state, and a negated state must be detected if the input signal reaches 1.9 V. In the SCSI-2 and SPI standards, any point between 0.8 V and 2.0 V could be used as the timing measurement.

Sample UltraSCSI Signals

Numerous variations on the details of the signals can be produced in UltraSCSI configurations. This section shows two types of signals as representative examples that validate UltraSCSI as viable under certain conditions. The first case explores a configuration that actually exceeds the recommended specifications. This is a complex cabled environment with a cluster of loads on one end and some distributed loads on the other end. The second case shows the signals over a 25-meter, single-ended point-to-point bus.

Complex Loads Figure 5 specifies a complex configuration and the single-ended SCSI signals that result at various positions along the bus. The logic

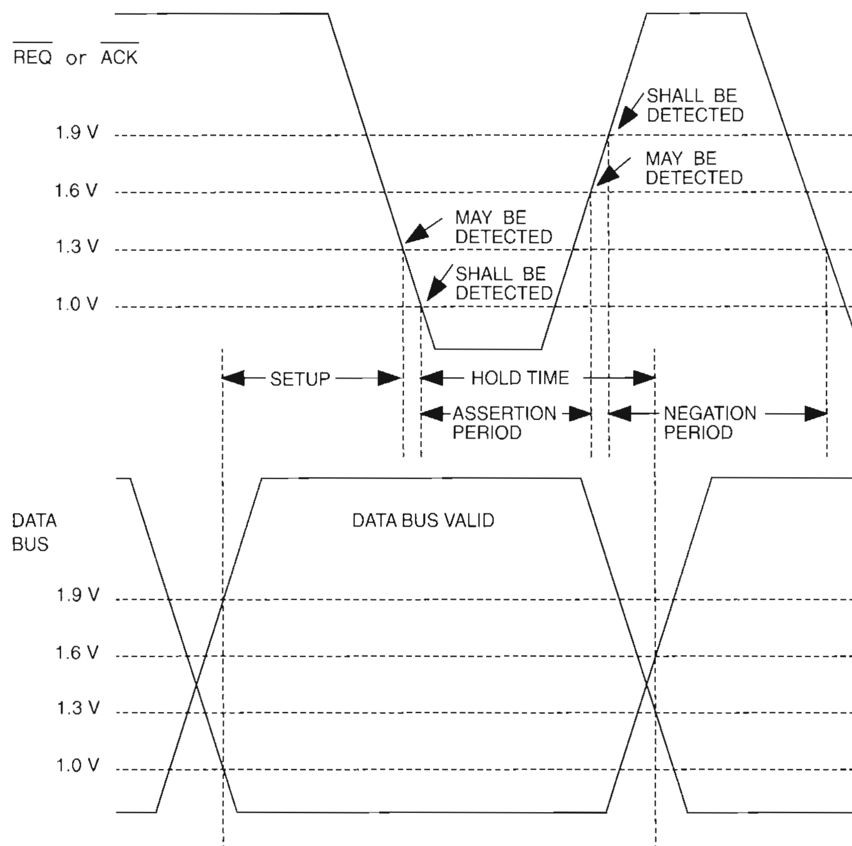


Figure 4
Single-ended UltraSCSI Timing Measurements

signal that is driving the SCSI driver chip is the first trace at the top; it provides a common timing reference for all the signals. The weakest signal is at device position 4, just after a relatively long run with no loads. This signal is below the 1-V level but has a very slow assertion slew rate that causes considerable loss of asserted state pulse width. This complex configuration works with the receivers used but does not have the timing margin required by the Fast-20 standard.

By varying the position of the loads so that there are no loads between the driver and the first load (not shown), the signal at the first load device is degraded even more than at position 4 in Figure 5. This is one reason that the overall length of single-ended UltraSCSI with many loads is restricted to 1.5 meters and that the total number of loads is limited to 8.[1] UltraSCSI devices connected to backplanes may be especially sensitive to attached cables that extend the total bus length more than 6 to 8 centimeters (cm) beyond the backplane. This reduced bus length is rather severe when compared to that allowed at the maximum fast SCSI transfer rate (a total of 3 meters).[4] In the section Small, Improved Interconnect, we show how to overcome this 1.5-meter, 8-device limit by using an active SCSI interconnect.

Applying the timing measurement methods shown in Figure 4 to the waveforms in Figure 5 illustrates that more careful timing specification methods do indeed help significantly to keep the timing margin high enough to use.

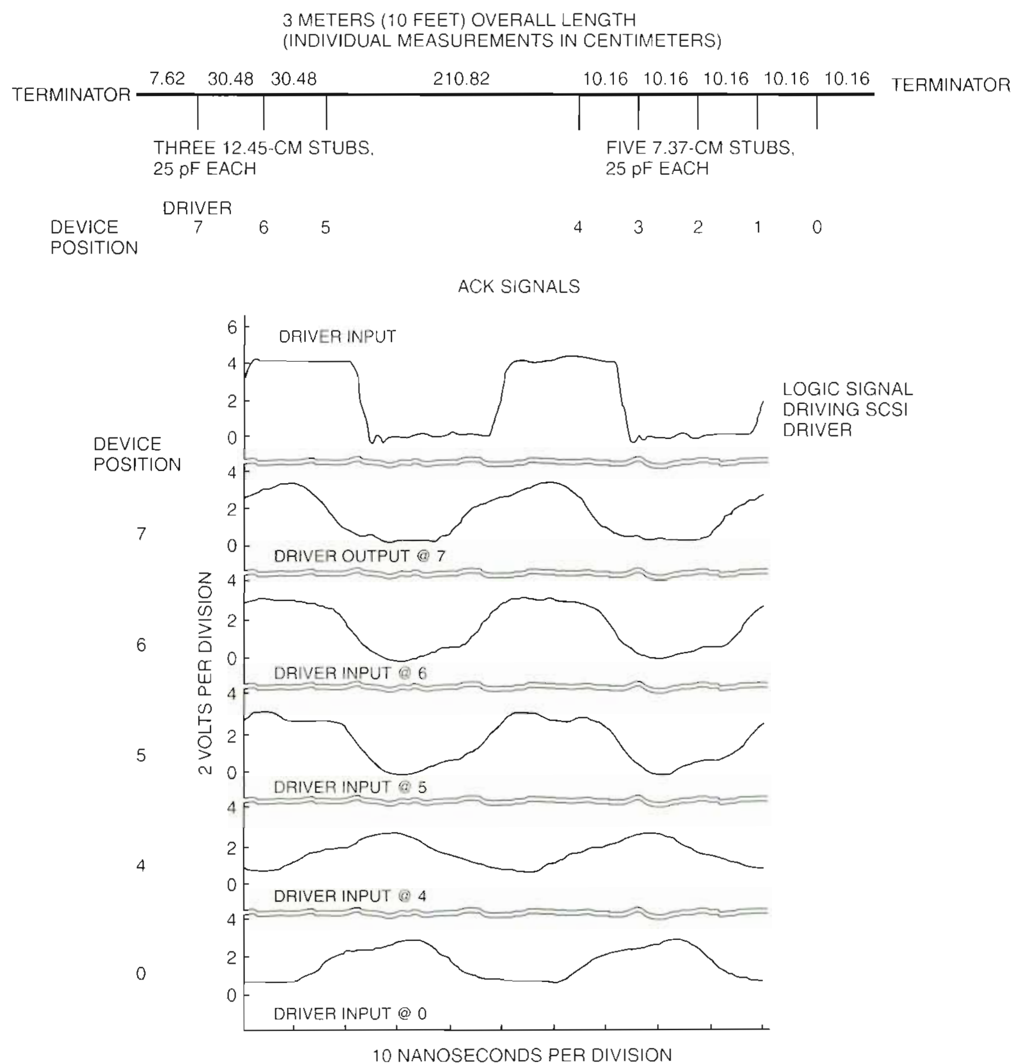


Figure 5
UltraSCSI Signals in a Complex Bus

Point-to-point Configuration If loads are present only at the ends of the bus, the transmission line between SCSI devices improves electrically. This occurs simply because the loads significantly disrupt the characteristic impedance and cause reflections and attenuation. The point-to-point signal at 25 meters has better amplitude and timing margins than signals in much shorter buses with closely spaced loads. Figure 6 shows a typical example of a point-to-point UltraSCSI signal. The format used in Figure 6 is the same format used in Figure 5.

Differential UltraSCSI

Differential UltraSCSI uses the same configuration rules as fast SCSI (25-meter total length, 20-cm [8-inch] stubs, 16-device load)[1] and uses the same timing values as single-ended UltraSCSI. The larger signal amplitudes and the common mode rejection property of differential transmissions help overcome the transmission line weaknesses in heavily loaded and long buses. As with any high-voltage differential system the costs—in terms of money, power, and space—are higher.

Other Requirements for UltraSCSI

The Fast-20 standard[1] contains a number of detailed requirements on the components used in UltraSCSI configurations. Included are slight modifications to the cable impedance, active negation requirements for drivers, special length limits for certain loading conditions, restrictions regarding the kinds of single-ended terminators to use, and timing budgets.

Summary of Developments in the Area of Increased Synchronous Data Phase Speed

The UltraSCSI (Fast-20) speed increase can be attributed to a systematic examination of the margins present in actual SCSI hardware and to the elimination of the excess margins. Advances in the integrated circuit industry enabled silicon designs to be specified with tighter controls on the driver and receiver timing and threshold properties than were possible when the SCSI-2 or SPI standards were developed. All the important changes needed for SCSI devices are contained in the silicon designs for the drivers and receivers. As a result, the user sees no difference between the appearance of UltraSCSI and that of ordinary SCSI.

The system integrator must use a more restrictive set of configuration rules than required for fast and slow SCSI. Also, the only impact on software is the addition of a new speed agreement code for the rates uniquely supported by UltraSCSI. This negotiation is done precisely the same way for UltraSCSI as for any other form of SCSI. Finally, UltraSCSI devices are 100 percent backward compatible with fast and slow devices. Although a device may be capable of the maximum UltraSCSI rate, it may be needed in a configuration that does not support UltraSCSI. In such a case, the UltraSCSI device would be used in the fast or slow mode and would have more margin at those slower speeds than it would if it were not UltraSCSI capable.

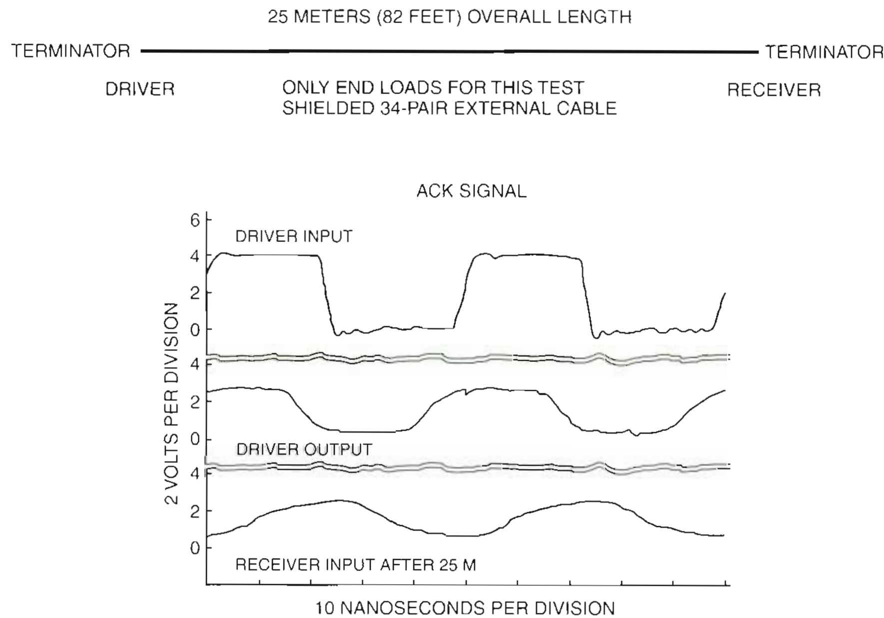


Figure 6
Point-to-point UltraSCSI Signals

Bus Expanders

As noted previously in the discussion of complex loads, there are rather severe limits on the configurations that can be achieved with single-ended UltraSCSI when implemented in a single bus. The extension to parallel SCSI architecture that overcomes this constraint involves using active circuits that connect SCSI buses electrically but isolating them from each other in a transmission line sense. These circuits have the general name expanders, since they expand the configuration capabilities of parallel SCSI.

Each individual bus has two terminators and its own transmission mode (single ended or differential) and obeys transmission line–based configuration rules as if it were the only bus in the system. When used with expanders, these individual buses are called bus segments. The collection of SCSI devices in all the bus segments that are electrically connected together is called the SCSI domain. One example of a SCSI domain using expanders is shown in Figure 7. Note that when using expanders, it is possible to have bus segments that do not have any SCSI initiators or targets but only serve to form an electrical interconnect between other bus segments.

Expander Properties

Expanders are available in two basic types: simple and bridging. Bridging expanders behave as a SCSI initiator or target, whereas simple expanders have a set of properties that make them look like a piece of wire with delay to the protocol. Simple expanders

- Cannot initiate SCSI IDs and arbitrations and cannot originate messages, although the expanders can read messages sent from initiators and targets
- Allow minimal arbitration propagation delay
- Yield a retransmitted signal timing skew (both delay and high/low) no worse than from valid SCSI initiators or targets
- Do not interfere with the REQ/ACK offset count
- Allow min/max pulse widths to be maintained
- Require the filtering of the SCSI RESET line
- Allow arbitrary placement of the initiator and the targets
- Require that terminator power not be connected between the segments being coupled
- Do not need to know the negotiated data phase speed or any other variable property of a transaction
- Require that there be no electrical or logical connection of the DIFFSENS line (a single-ended signal that indicates the transmission mode being used on the bus segment) between segments being coupled
- Issue a SCSI bus RESET signal on one segment on detecting transmission mode (single-ended/LVD, etc.) changes on the other segment

Simple expanders are becoming available from several sources in the industry for use with UltraSCSI.

Domain Rules Using Simple Expanders

When using only simple expanders in a domain, six rules must be observed:

1. All bus segments in the domain must comply with their individual bus segment length limits and other segment-related requirements.
2. Any segment between two other segments must support the highest performance level that can be negotiated between the two other segments. For example, two wide UltraSCSI segments must not be separated by a segment that does not support both wide SCSI and UltraSCSI.
3. The maximum propagation delay between any two devices in the domain cannot exceed 400 ns. A special case exists for devices that use extremely long times for responding to BUS FREE (the so-called BUS SET DELAY)—the one-way propagation limit is 300 ns instead of 400 ns.
4. The number of addressable devices cannot exceed 16 unless the domain contains bridging expanders.
5. A branch/leaf architecture must be observed; loops are not allowed.
6. The REQ/ACK offset negotiated between any two devices must be large enough to ensure that adequate offset and buffering is available to accommodate the round-trip time between the devices. For the maximum UltraSCSI rate with a 400 ns maximum one-way domain propagation time, the

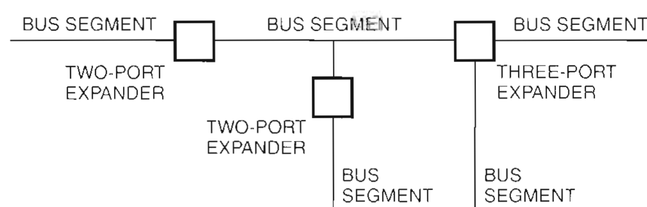


Figure 7
SCSI Domain Built Using Expanders

minimum offset is 18. (This offset level is derived by considering a maximum round-trip time of 800 ns at 50 ns per transfer [800/50 = 16] and somewhat arbitrarily adding two transfers to account for some additional delay due to the processing time in the silicon.)

Achieving the 400-ns one-way domain delay requires expanders that will not pass the wired-OR glitch (noted earlier in the introduction) between bus segments. This filtering of the glitch allows the bus segments to settle individually.

The propagation delay through an expander directly subtracts from the physical distance between devices. It is therefore desirable to use expanders with small delays. For a single-ended–to–single-ended application, the delay can be as low as 10 ns. For a single-ended–to–differential application, the delay is typically around 100 ns, which is another significant penalty to using differential bus segments.

More detail concerning these rules and other properties is available in the draft ANSI document: *SCSI Enhanced Parallel Interface,*[5] which was edited by the author of this paper.

Summary of Improvements Related to Bus Expanders

The use of simple expanders dramatically extends the utility of single-ended UltraSCSI. The most obvious example is the ability to introduce point-to-point segments where additional length is needed. A less obvious example is the ability to create star or hub configurations by clustering simple expanders into a local physical area. An example of a three-port SCSI hub is shown in Figure 7. Note the three simple expander circuits internally connected within the hub. Simple expanders also make it possible to mix single-ended and differential SCSI devices in the same domain, to achieve the full 16-device count, to add and remove bus segments without shutting down the entire domain, and to achieve differential performance without incurring the extra cost of differential. Bridging expanders offer the same transmission isolation as simple expanders and may allow increasing the number of devices in the domain to as high as 946,[5] but bridging expanders are not as well developed as simple expanders and will not be explored in depth in this paper.

Note that the improvement in signal integrity is dramatic when using expanders with backplane applications. Therefore, it is good practice to use an expander whenever connecting a SCSI cable to a backplane that contains SCSI devices.

Smaller, Improved Interconnect

Another recent development in parallel SCSI technology is the introduction of much smaller external physical interconnects and more capable internal device interconnects. The SCSI connectors and shielded cables have historically been large, bulky, and generally difficult to manage.

Spearheaded by activities that began in 1995 in the SFF (formerly Small Form Factor) industry group, standardization is under way of two new connector families that offer unprecedented levels of functionality and true multisourcing of complete connectors for parallel SCSI. These families are the Very High Density Cabled Interconnect (VHDCI)[2] shielded connectors that reduce the overall size of an external connector by two thirds and the Single Connector Attachment–2 (SCA-2)[7] unshielded connectors that integrate into a single connector all the functions needed to run a peripheral. The VHDCI family revolutionizes the external SCSI interconnect and the controller parts of the internal SCSI interconnect; the SCA-2 family does the same for the internal device interface.

For the first time, complete connectors—not just the mating interface—are being standardized. This feature is essential to achieving interchangeability and second sourcing for connectors with the same style of termination-side contact. The VHDCI family is specified in 26 different forms, all with exactly the same mating interface, so that virtually any kind of device or cable assembly design can be accommodated. Interestingly, this array of choices for the connectors does not increase the complexity of the interconnect but rather opens up new ways for product developers to design products while maintaining a simple and physically interoperable separable connector interface. In fact, this ability to accommodate a variety of product design requirements without changing the separable interface is one reason that SCSI is becoming *less* complicated.

Similarly, the family of SCA-2 connectors for SCSI internal devices and cables is following the VHDCI standardization model, with a significant number of intermatable forms being standardized. These connectors offer the ability to bring all the SCSI signals, all the power and ground connections, and all the optional signals, such as IDs, spindle sync, and power fail, out of the device through a single unshielded connector. This feature dramatically shrinks the cost and complexity of interconnecting an array of SCSI devices.

Using an SCA-2 connector, the device may be inserted into a backplane without using cables. If the SCA-2 and backplane combination is not used, a SCSI cable (50-pin or 68-pin conductor), a four-lead power cable for ground and power (5-V and 12-V), and one or more smaller cables for the IDs etc., are required for *every* device in the system. Each of these cables is routed differently, has different current carrying and other electrical requirements, and has very different connectors. Although this cabled option is flexible and offers significant advantages in some systems, it is usually not the best solution in the device array and modular packaging applications that are required for the

higher-end applications. Therefore, the SCA-2 is a significant factor in the dramatic reduction in complexity of higher-end SCSI device applications.

VHDCI Connectors

The physical size of the VHDCI connectors is much smaller than the earlier versions, as seen in Figure 8. Because of its low profile, the VHDCI 68-pin family is approximately half the height and twice the width of the latest Fibre Channel external connector, the High-Speed Serial Data Connector (HSSDC). Figure 9 shows a comparison of the VHDCI and HSSDC connectors. The same panel space is required for either technology.

The VHDCI connectors shown in Figure 9 are closely spaced, but the orientation of the polarizing shield connection is 180 degrees different between the upper and lower connectors. This arrangement allows an offset cable assembly to be used where one side is flat. This same cable assembly may be used on both the upper and lower connectors without interference. The specifications of the VHDCI interface ensure that neighboring PC option slots will not have interference even if all the SCSI ports have cable assemblies attached.

The VHDCI connector is useful for multiport applications such as RAID (redundant array of inexpensive disks) controllers. Figure 10 shows examples in which the wide version of the connector family has allowed at least a doubling of the number of ports possible in a single controller form factor. As illustrated in Figure 10, the device design enables up to four wide SCSI ports on a single PC option card cutout.

The VHDCI retention scheme is also significantly simplified by introducing a three-way retention post for the bulkhead connector. This post accepts (1) the conventional jackscrews, (2) a squeeze-to-release clip for positive retention with rapid release, or (3) a detent ring retention that requires a stronger pull than that required with no retention but no action other than

SCSI-1 LOW-DENSITY
NARROW (50 PINS)

SCSI-3 HIGH-DENSITY
WIDE OR NARROW (68 PINS)

VHDCI WIDE OR
NARROW (68 PINS)

VHDCI NARROW
(36–40 PINS) MICRO SCSI

Figure 8
External SCSI Connectors

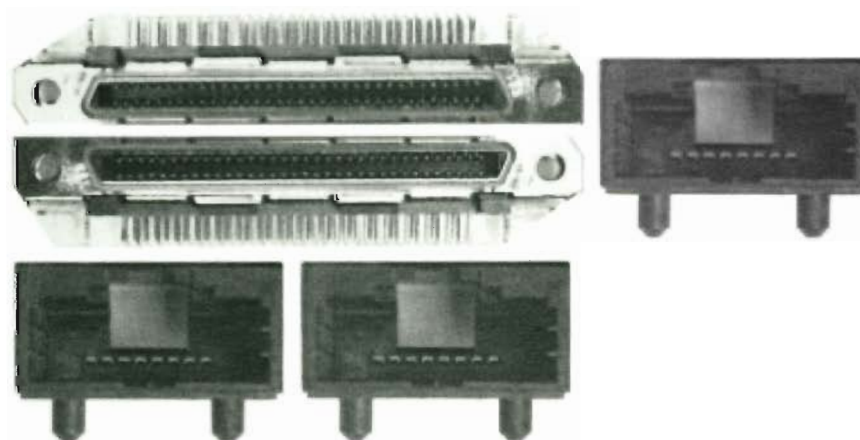


Figure 9
Comparison of the 68-Pin VHDCI and Fibre Channel HSSDC Connectors

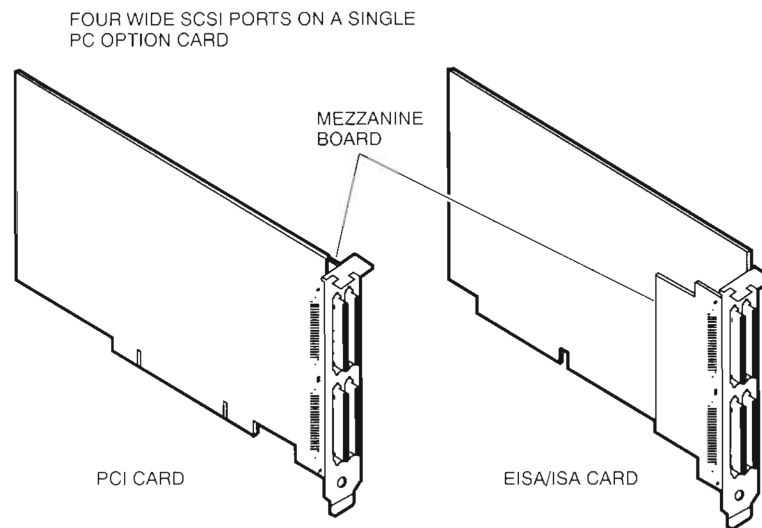


Figure 10
Four Wide SCSI Ports on a Single PC Option Card

pulling or pushing. The choice of retention type is made in the cable assembly. All 68-pin VHDCI cable assemblies that comply with the SFF specifications work on all 68-pin VHDCI mating connectors.

Figure 11 shows the details of the 68-pin VHDCI system. The lip in the jack post provides the securing point for squeeze-to-release clips and for split-ring detent retention. The center of the jack post is threaded for use with jackscrews.

Although smaller than the high-density connector, the VHDCI connector is durable. It has no pins that can bend; its retention scheme uses the same-size jackscrew thread as the high-density wide connector; and its contacts are imbedded in the housing where they cannot move or become distorted.

SCA-2 Connectors

The SCA family uses an 80-position, leaf-style contact to interface all active SCSI lines, three power voltages, and device control signals. This connector is considerably smaller than the collection of the three different connectors used for power, options, and SCSI bus in a cabled system. There are two basic versions of SCA connectors: SCA-1 and SCA-2. Both versions are unshielded and useful only within shielded enclosures. The SCA-1 has 80 positions with all contacts designed

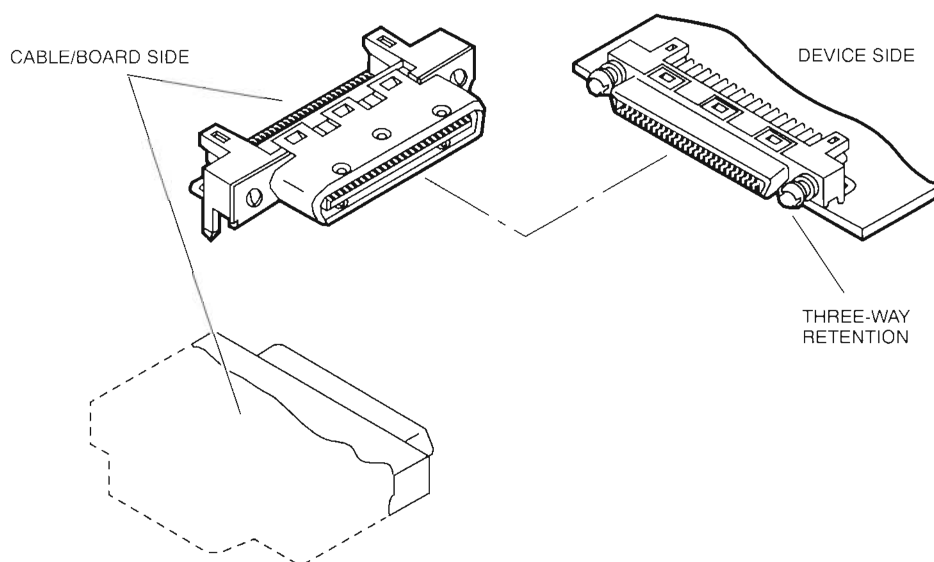


Figure 11
Overall View of the 68-pin VHDCI System

to be the same length. The SCA-2 can be mated to the SCA-1 but has advanced grounding contacts and sequenced signal and power contacts for supporting hot plugging and blind mating (no visual feedback during mating). Both versions are available in many styles, which differ by the termination-side structure and overall orientations.

The SCA-1 is not a documented standard and is being replaced by the SCA-2. The SCA-2 connector was introduced to SFF in 1995[7] as the first step toward formal standardization.

Two special features exist in the SCA-2 connector. First, two contacts, one on each side of the connector, provide the first make/last break for the ground connection. This design ensures that a common electrical ground is established between the device and the system before any power or signal connections are made on device insertion. Upon removal, these contacts ensure that the ground stays intact throughout the disengagement of the signal and power pins.

The second feature allows the special long power contacts to precharge bypass capacitors before the main power contacts make. This reduces the disturbance to the power distribution system and eliminates any arcing on the service power pins. Two pins at the extreme ends of the connector indicate that the connector is fully mated. The overall view of the SCA-2 system is shown in Figure 12.

The size of the connectors in the SCA family has not decreased dramatically. The connectors need to maintain enough size to achieve blind mating alignment, and, for backplane applications, there is little advantage in having a connector that is smaller than the device. With 89-mm (3.5-inch [in]) or the newly proposed 76-mm (3-in) form factor devices, the SCA connector comfortably fits within the device boundaries.

The use of backplanes for direct device attachment is possible because all the electrical connections for the device are available in one connector on the device. This design eliminates the cables used to attach the device and the space required for the connectors, thus significantly shrinking the size required to package multiple devices.

External SCSI Cable

The external cable for SCSI is shrinking also, through the use of smaller-gauge wire, better dielectrics, and less jacketing material, as illustrated in Figure 13. Formerly, wide SCSI required a cable of approximately 12.70 mm (0.50 in) in diameter (a 126.677-mm² [0.196-in²] cross section) with 28-gauge wire. Today, wide SCSI cables with 30-gauge wire are shipping with diameters of 9.40 mm (0.37 in) (69.398-mm² [0.107-in²] cross sections). Cables with 7.62-mm (0.30-in) diameters (45.61-mm² [0.07-in²] cross sections) are possible with 32-gauge wire and inexpensive dielectrics for wide SCSI. Cables with 6.35-mm (0.25-in) diameters (4.987-mm² [0.049-in²] cross sections) for narrow SCSI (45.61-mm² [0.07-in²] cross sections) are flexible and manageable—similar in size and flexibility to a desktop computer power cord and smaller than many serial cables. When used with active single-ended, LVD, or HVD terminators, the 32-gauge wire is adequate for distributing terminator power and SCSI signals in most applications. Long cables should not be used for terminator power distribution.

Further reductions in the connector and cable sizes need to be weighed against the ease of handling, the need for sufficient strength to survive normal service stresses, and the cost increases at very small sizes. The combination of the VHDCI connector and 30/32-gauge wire sizes is a good optimization.

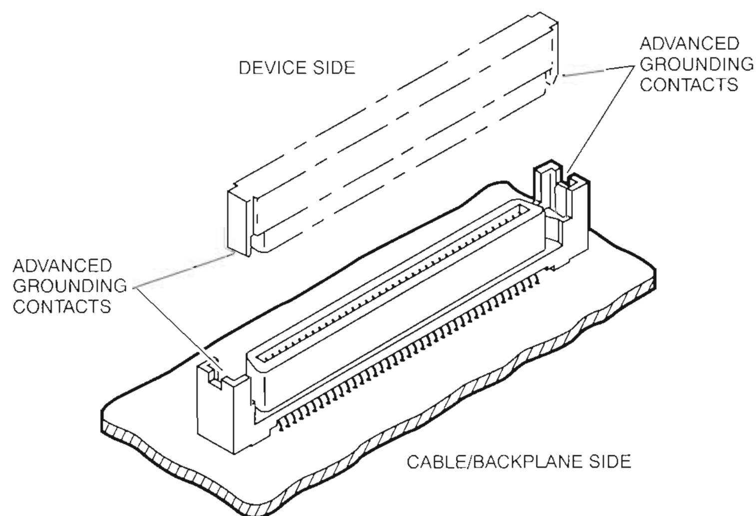


Figure 12
Overall View of the SCA-2 Connector System

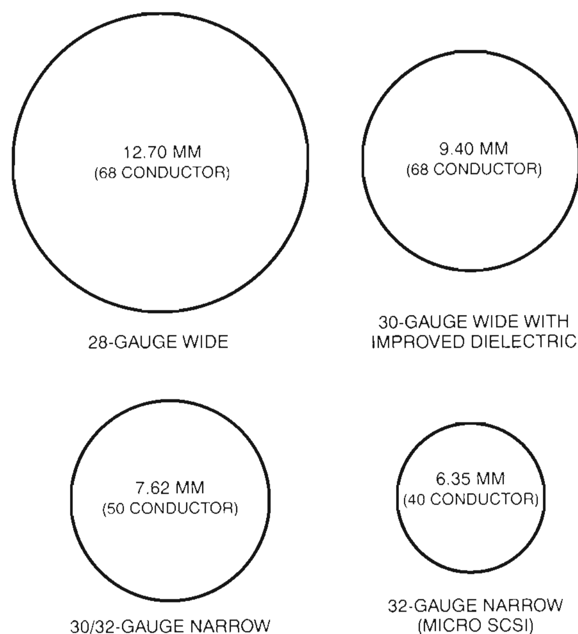


Figure 13
External SCSI Cable Diameters

Summary of the Benefits Derived from a Smaller, Improved Interconnect

The VHDCI connector and smaller cables combine to offer a robust yet user-friendly revolution in SCSI interconnect. The leaf-style contact of the SCA connector eliminates problems with bent pins that frequently bedevil the older wide SCSI connector. The ability to use up to four wide UltraSCSI ports in a single PCI option slot increases the SCSI connectivity per PCI slot to 60 devices (from 15 devices). By using multiple PCI slots, hundreds of SCSI devices can be connected to a single PC or workstation. In addition, the SCA-2 connector implements the essential contact sequencing required to perform SCSI device hot plugging.

Device Insertion and Removal Bus Transients

The multidrop feature of the SCSI bus allows device removal and replacement without disturbing the communications between other SCSI devices, if the electrical disturbances caused by the device being added or removed are not detected by any other SCSI devices. Thus, it is architecturally possible to dynamically reconfigure the device population without interrupting existing data transmission processes between operational devices.

The transients involved with device insertion and removal include mechanical vibrations, power distribution instabilities, SCSI terminator power noise, electrostatic discharge (radiation and induced current), and SCSI signal line noise. All except the SCSI signal line noise and the terminator power noise are handled by the storage system design and therefore are not directly part of the advancements in parallel SCSI. The SCSI terminator power noise is determined by the size of the decoupling used on the SCSI terminators and the size of the capacitance on the device being inserted. This noise is easily controlled by ensuring that these sizes meet the values specified in the SPI standard.[4]

The delicate case is when the SCSI signal lines are involved, which is the subject of this section. To determine the nature and magnitude of these signal line disturbances, one must understand the following three mechanisms: (1) the overall sequence of events, (2) the electrical dynamics of connector contacts when used in the SCSI application, and (3) the electrical consequences on the bus when the device makes/breaks contact with the SCSI signal line.

There are two sequences of interest: insertion and removal. The removal process is easy to grasp after the insertion process is understood.

Single-ended Device Insertion

The initial conditions considered for SCSI device insertion assume a SCSI device with its ground solidly and continuously connected to the ground of the SCSI bus. This connection is easily accomplished, for example, by using sequenced contacts where the device ground makes connection well before any signal connection. In this state, the SCSI device pins present a maximum fully discharged capacitance of approximately 25 picofarads (pF). After the device signal pin contacts the bus, this capacitance becomes charged (by extracting charge from the bus) to the voltage on the signal line at the time of the insertion.

These values range from approximately 3 V for negated lines to nearly 0 V for asserted lines.

Since the SCSI device being inserted is logically off (i.e., there is no driver current), the only current that needs to flow is that required to charge the 25-pF capacitance. This is sharply different from many connections in electronics in which current flows through the contact after an electrical contact has been established.

In the case where no bus voltage changes occur except as a result of the device insertion, the insertion transient begins with the initial contact and ends when there is no further bus voltage change with time (the steady state voltage). Once the device pin voltage reaches the steady state bus voltage, no further current flows through the contact.

Therefore, once the device capacitance becomes charged to the steady state signal line voltage, no further disturbances to the signal line voltage will occur even if the contact opens momentarily during a chattering event. The voltage on the device capacitance changes during the transient from a discharged state (zero voltage) to the steady state signal line voltage, with the current always flowing into the device capacitance.

If the signal line voltage changes after the insertion transient is completed (because of events such as being driven by other devices, by noise, or by the inserting device beginning to use its own driver), then current will again begin to flow through the contact. This is a normal SCSI condition for contacts in service. If the signal line voltage changes during the insertion transient because of events other than the connector contact effects (e.g., signals changing because of being driven by other devices, other noise), then it is more difficult to determine exactly where the insertion transient ends. The beginning of the insertion transient will still be marked by a charging of the device capacitance. Examples presented later in this paper show both insertion events and driving events from other devices occurring at the same time.

The time required for complete contact mating on all SCSI signals in the bus is up to six orders of magnitude greater than the time required for a SCSI signal to change state. Therefore, signal level changes are likely during the insertion process. The electrical behavior of the contact as it continues wiping (sliding after initial contact is made) from its initial contact point to its final resting position becomes a critical part of the process. The following subsections explore this behavior in detail.

Connector Insertion Dynamics The data presented in this section were derived from a DIGITAL DSSI bus in 1990. The DSSI bus is nearly identical to the SCSI bus, and many of the results apply without modification to SCSI. Similar data have been observed on the SCSI bus, but the complete set of data presented in this paper is not presently available from actual SCSI hardware. The disturbances in the DSSI bus are larger than those seen in the SCSI bus, because the DSSI voltages are slightly higher (3.5 V for DSSI compared to 2.8 V for single-ended SCSI), and the instrumentation capacitance (~10 pF) adds significantly to the device capacitance because of the state of the art for scope probing in 1990. Numerous tests with modern scope probes (0.6 pF or less) of SCSI hardware have shown that the SCSI disturbances are indeed qualitatively the same but significantly less in size than those shown here from the DSSI hardware.

The mechanisms described apply to any system in which the insertion transient is caused by the charging of a small capacitance. Figure 14 shows the basic test setup. A device is inserted into a connector with scope probes attached on either side of the mating interface and with an additional probe attached to the bus some distance from the connector. The voltage on the device side of the connector is used as the trigger signal into a digital storage scope so that the events before, during, and after the mating event can be examined. This is clearly a single-event type of measurement, so a high sampling rate (1 billion samples per second) and significant scope memory is required to capture the waveforms. The scope probes used have a 1-megohm input resistance.

The connector used for the tests in this section has multiple parallel pins that all mate and demate in the same general time period. There is no intentional difference in the pin lengths. The time relationship between the mating events on two neighboring pins was explored first. By choosing neighboring pins, the differences between the pins is kept to a minimum so the time differences observed should represent the best pin-to-pin synchronization in a mating event.

For this test, a probe was attached to each of two pins, and the connector alone (not part of a device) was mated to the bus segment connector. Figure 15 shows the results.

Both pins appear to show instantaneous transitions between the charged and discharged states on the time scale that was required to capture both events on the same plot. The mating events are separated by approximately 19 milliseconds, and there is no evidence of any discharging after the initial charging has occurred. Since the scope probes have a 1-megohm input resistance, any lack of contact during the wipe portion of the mating will allow the capacitance to discharge through the probe with a time constant of approximately RC where R is the scope probe resistance and C is the sum of the connector pin and probe capacitances. Assuming a total of 10 pF, this gives a decay time constant of 10 microseconds.

Figure 16 shows another mating event on pin 1 at a 500 times more sensitive time scale. In this case, some evidence of momentary opens is seen with the

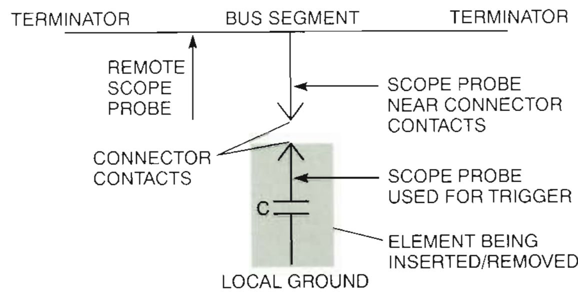


Figure 14
Test Setup for Insertion/Removal Transients

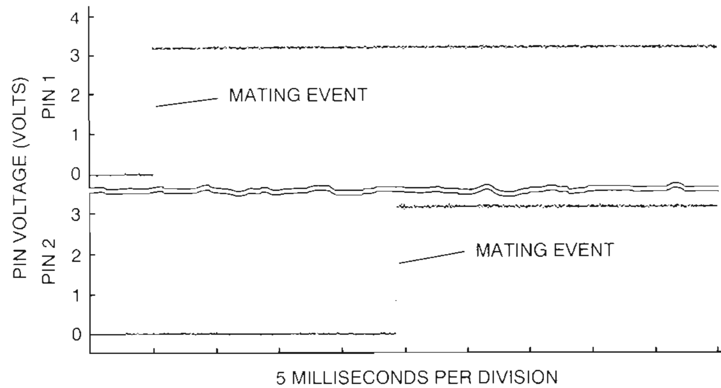


Figure 15
Time Relationship between Mating Events for Two Pins in the Same Connector

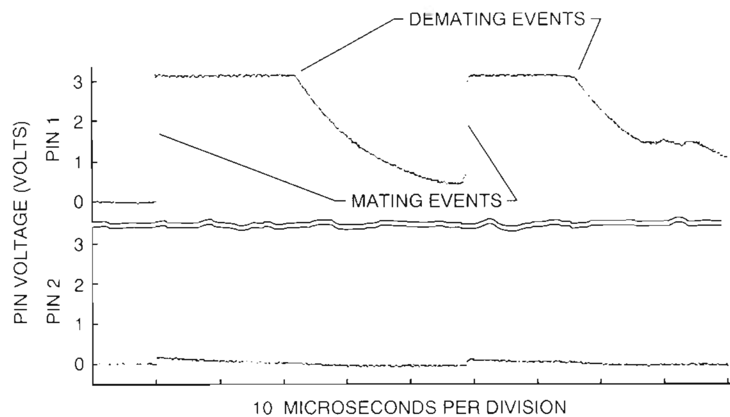


Figure 16
Contact Bounce Events

expected decay dynamics. The actual time constant is a bit longer than 10 microseconds because of some capacitance in the connector pin. This bounce behavior may or may not be present during the initial stages of the event shown in Figure 15, but clearly the behavior is not visible in the figure. To observe the suite of transients that exist in the mating process, one must examine the transients at several different time scales. In general, this requires repeating the mating events, since the dynamic range of the scopes used was insufficient to capture all the detail in a single event.

The initial mating event on pin 1 still appears to be instantaneous on the time scale used in Figure 16, but some slope is visible in the second bounce event. Also, during the second decay period, a shelf in the decay indicates that a partial, high-resistance contact was briefly experienced. Pin 2 is not close to making a contact at the time range shown in Figure 16. The figure shows a small amount of cross talk in the pin 2 voltage waveform caused by the pin 1 transients.

This data clearly shows that the details of the mating process are highly complicated and intrinsically

unpredictable. Therefore, the best we can hope for is to establish some limiting cases for the important parameters. The limiting features shown in Figure 16 are the extremely rapid initial mating event and the decay times. We examine these rapid transients in detail later in this section. The decay times are determined by the actual contact resistance and the resistance of the leakage path to local ground. For normal SCSI devices, there is very little leakage to ground on the device pin so the opens produced by the bounce have no effect.

Some cases observed indicate much more complex bounce structures. Figure 17 shows a case in which the mating connection is not established until more than 700 microseconds have passed.

The data in Figures 15 through 17 were all acquired from the same connector contact during separate mating processes. Typically, the details of the mating event are very different even under nominally identical conditions.

Another type of mating event is shown in Figure 18. This event requires approximately 10 microseconds to make the transition from uncharged to charged, and there is no bounce. This particular event produces almost no cross talk into pin 2. Events with these characteristics are somewhat rare and are called gradual transients in this paper.

Figure 19 shows a closer look at the rapid transient type of mating event. In this figure, we have added a device capacitance of approximately 20 pF to the scope probe for a total of approximately 30 pF. Notice that the transient requires 2 to 3 ns to substantially complete its charging. There is a ratio of nearly 10^7 between the mating events on different pins in the same connector and the rapid transient of a single contact.

Limiting Parameters for the Rapid Transient The question of whether the rapid transient shown in Figure 19 is the worst case needs to be explored because the duration of the transient affects the disturbance on the bus. Some bounding features and some implications of the observed behavior of the rapid transient are noted in this subsection.

Assuming that the transient event occurs in 2 ns and that the velocity of impingement just prior to the first mating event is 1 meter per second, then the distance traveled by the contact would be 2 nanometers (nm).

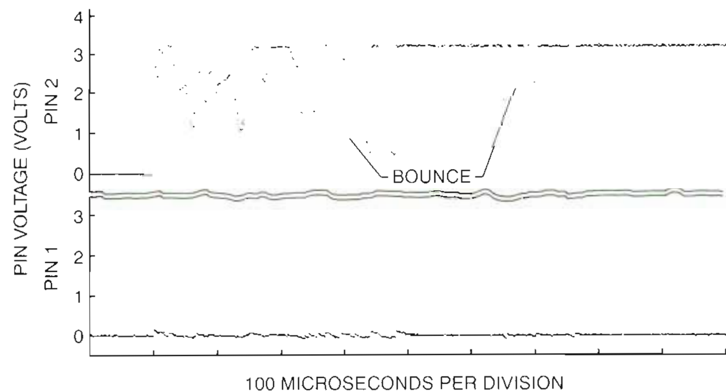


Figure 17
Extended Mating Bounce Events

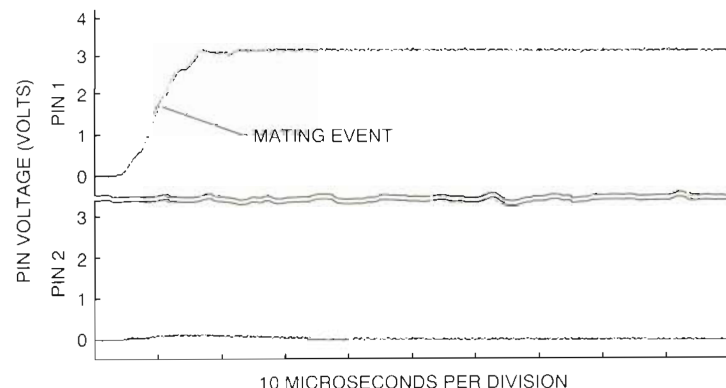


Figure 18
Gradual Mating Event, No Bounce

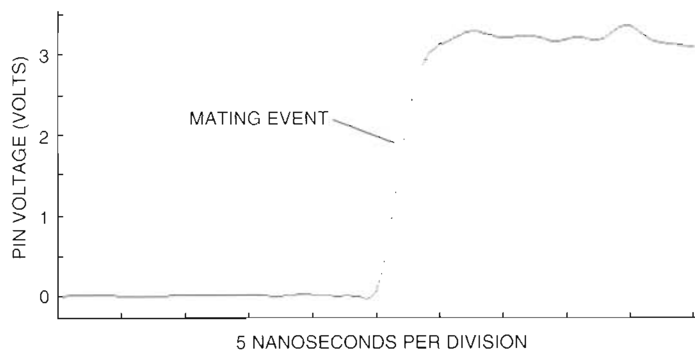


Figure 19
Detailed Structure of the Rapid Transient

This distance is equivalent to a few atomic distances. The distance traveled during the gradual transient shown in Figure 18 would be approximately 10 microns, and during the extended bouncy case shown in Figure 17, approximately 1 mm. The velocity for the latter two cases would likely be somewhat reduced because of the mechanical interference between the pins, and the actual distance traveled is probably significantly less. There is little opportunity, however, for the velocity to be reduced for the rapid transient, and this distance of 2 nm is probably at least the correct order of magnitude.

The following calculation shows the total current levels required to charge the capacitance in 2 ns.

$$Q = CV = 30 \times 10^{-12} \text{ pF} \times 3.5 \text{ V} = 10.5 \times 10^{-11} \text{ coulombs},$$

where Q represents the total charge, C is the capacitance, and V is the voltage. Since this charge is transferred in a time t of 2 ns, the average current is

$$Q/t = 10.5 \times 10^{-11} \text{ coulombs}/(2 \times 10^{-9} \text{ ns}) = 52.5 \text{ milliamperes (mA)}.$$

For a gradual transient that takes 10 microseconds, the average current is approximately 10 microamps. These calculations show that the most severe amplitude disruption to the signal on the bus occurs with the rapid transients, since relatively large current must be supplied in a short time to charge the capacitor.

The next item to be examined is the current density that must exist during the transient. Since the contacts move only 2 nm and the surface finish of actual contacts is not nearly this smooth, it is reasonable to assume a square 2-nm contact. Clearly, this assumption is not rigorously defensible and could be the subject of an entire study area in its own right; however, there is no basis for assuming that the lateral contact region would be any different than the contact area in the mating direction. The basic conclusions would not be affected even if we assumed a hundredfold lateral increase in contact area. Attempts to use scanning electron microscopy to examine the actual contact area were not fruitful in establishing the actual initial physical contact area because of the severe physical disruption that occurs on the microscopic level and because of the small sizes involved.

Under these assumptions, the physical contact area is assumed to be $(2 \text{ nm})^2$ or 4×10^{-14} cm^2 in the following calculations. The current density to support the 50-mA rapid transient current is therefore approximately 10^{12} A/cm^2. Typical current densities in copper and other metals are less than 10^6 A/cm^2. The electromigration onset current is of this same order. The current density in the rapid SCSI transient is a million times greater than that which metal can support.

To support the massive current density, the actual contact area must be much larger than the initial physical contact area assumed in the above calculations. The author believes that this can be explained by a micromolten metal-to-metal joint that is formed upon initial contact and that the front of the melt propagates (probably through phonon interaction) at approximately the speed of sound in the metal. This process would create crudely a thousandfold increase in the effective insertion velocity and would result in a millionfold increase in contact area, since the melt would propagate in all directions.

This mechanism would produce reasonable current densities and would form an intimate metal-to-metal interface with both contacts that would aid in reducing the contact resistance. The micromelt size becomes rapidly self-limiting, with the expanding contact area causing decreased current density, which in turn, causes decreased melt temperature.

As discussed in the next section, the actual contact resistance during the rapid transient cannot be large. If this resistance is large, as in the case of the gradual transient, the mating event is much less disruptive.

Many variations on the mating transients can be observed, but we do not attempt to show all of them

in this paper. One special variation, however, is worth noting—the combination of rapid and gradual transients in the same mating process. Sometimes the mating process starts with a gradual transient and then shifts to a rapid transient. Figure 20 shows a complex mating process in which (1) a gradual transient initiates, (2) a rapid transient starts but does not complete, (3) the rapid transient ends, (4) another gradual transient process starts, and (5) another rapid transient finishes the charging process.

This observation is consistent with several possible microprocesses during which the initial rapid transient extinguishes before completion.

- The micromelt becomes physically torn apart by the advancing motion of the contacts. (This process is unlikely because of the excessively slow physical motion.)
- The micromelt explodes. (This process is likely.)
- The micromelt becomes resistive through the contamination of the melt with insulating material.
- The micromelt front reaches a thin region and opens because of the lack of material.
- The micromelt front reaches an insulating region.

On further movement of the contacts, a new rapid transient condition is encountered between different metallic peaks of the contacts, and a new rapid transient begins. Figure 21 shows a conceptual representation of this process.

Gradual transients appear to be associated with normal current densities (i.e., 10^6 A/cm^2) and much higher contact resistance than rapid transients. In cases where a micromelt is not initiated, the low contact resistance associated with the liquid metal–to–solid metal interface and the expanded contact area are not present. Therefore, one way to eliminate the mating disturbance caused by the rapid transients is to ensure that a micromelting process is not possible.

In the process shown in Figure 20, it is probable that a gradual-type contact is being maintained somewhere else in the contact, since no voltage decay is evident when the rapid transient ends. Indeed, it is to be expected that the rapid transient mechanism would not operate after the capacitance is charged to a certain level, since there would not be enough energy difference to initiate and sustain a rapid transient. Therefore, the gradual transient is the behavior derived from an extrapolation of the normal mechanisms that produce contact resistance. This detailed discussion is pursued because we must understand the basic physical mechanisms to gain confidence that we are considering the worst-case disturbances.

Single-ended Device Removal

During the process of removal, the device pin separates from the bus. Since both the bus and the device are at the same voltage just before the separation, no current is flowing unless the bus voltage changes when the contact is in the process of separating. Therefore, in most cases the separation process causes no disturbance.

Bounce can occasionally be observed during the demating process when there is a leakage-to-ground path present on the device side. Of course, if a voltage decay occurs and the contacts re-connect, the mechanisms are essentially the same as for the insertion transient. The key point is that no additional mechanisms have been noted for device removal that could be more disruptive than those operating during the insertion process. In the limit, the removal process could produce as much disruption as the insertion process.

Figure 22 shows two examples of demating. The demating events shown in Figure 22a have only approximately a 60-microsecond separation. This separation is exceptionally small, and it is theoretically possible to have coincidental contact-to-contact events (within the precision of the instrumentation). The demating event with bounce shown in Figure 22b was acquired on exactly the

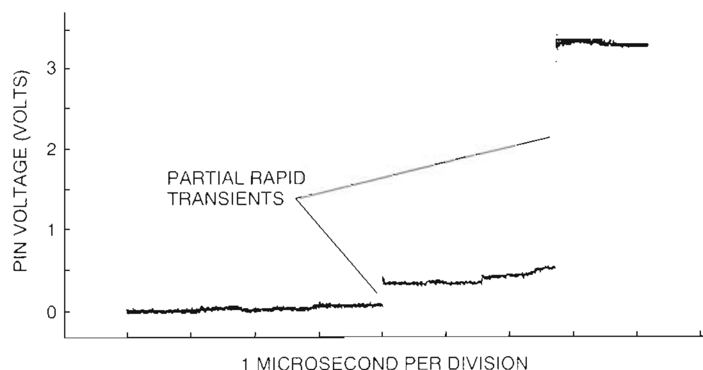


Figure 20
Gradual and Rapid Transients in the Same Mating Process

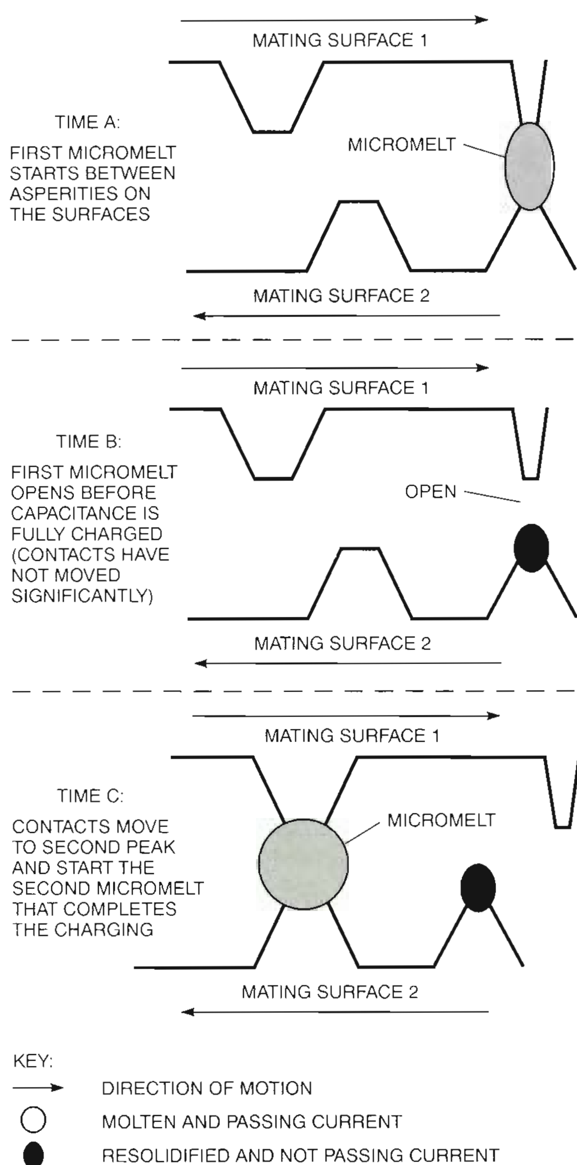


Figure 21
Architecture of Combination Gradual/Rapid Mating Event

same pins in exactly the same connector used for the events in the top of the figure, and there is no evidence of any activity on pin 2. Pin 2 demated long before any activity was seen on pin 1. Again, this underscores the unpredictability of the details of any given event.

Impact of Device Insertion and Removal on Bus Signals
This section contains several examples of the noise produced on the bus side of the connector. Actual devices with approximately 25 pF of capacitance were used to obtain the data. This capacitance value is increased by the probe capacitance. On the bus side, there is also some increased capacitance caused by the probe used to acquire the bus side signal. Figure 23 shows the basic impact of a rapid transient on the bus side of the connector and the time relationship of the bus disturbance to the voltage on the device side. The bus voltage is reduced while it supplies the necessary charge to the device pin. After the device capacitance is charged, the bus resumes its voltage level before the insertion transient (more or less).

In this test, the bus pulse is approximately 3-ns wide at its midpoint; its peak amplitude is approximately 1.25 V. This pulse is significantly larger in amplitude than that produced from a device alone.

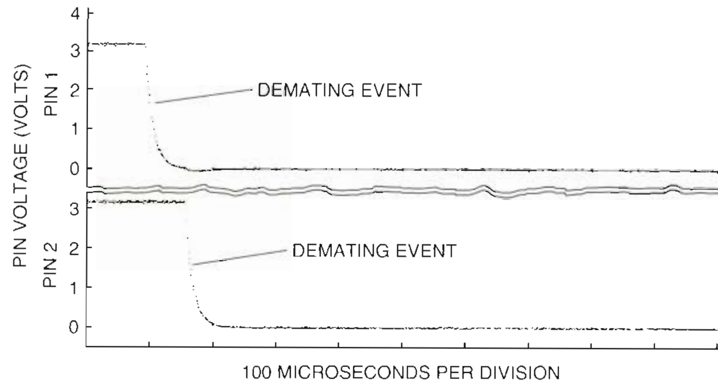
One of the more interesting features of the signals in Figure 23 is the lack of commonality or tracking in the signals after the rapid transient has passed. In the simplest interpretation, one would expect both sides of the connector to have nearly the same voltage (at the least to be within the accuracy of the 0.1-ns propagation time between the probes). The following discussion addresses the author's current thinking on the reasons for this lack of tracking.

Instrumentation effects, such as resonance or differences in probe properties, were ruled out by using both probes on the same signal and noting that there was little difference in the signals reported from each channel. Later, typically after a few microseconds, the voltages do become effectively the same.

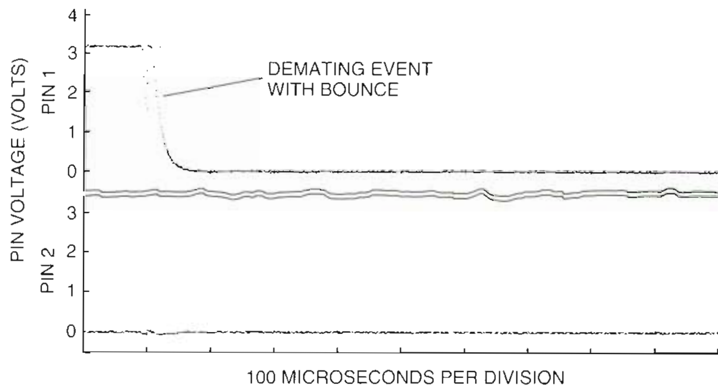
Because a significant voltage difference is present for relatively long times, there must be a significant voltage source between the contacts to support this observed difference. In the initial stages, the difference between the pin voltages is approximately 3 V. If the current is the one calculated in the section Limiting Parameters for the Rapid Transient, that is, approximately 50 mA, then the current-limiting impedance must be at least $3/0.05 = 60$ ohms. This impedance, coupled with the parasitic capacitances and inductances, serves to blunt the instantaneous electrical energy transfers that would be implied by a very low source impedance. If the source impedance were very low, then both sides would have to track shortly after the initial contact.

Part of this limiting impedance is the loaded or local transmission line impedance of the bus. The characteristic impedance is nominally approximately 100 ohms for an unloaded bus. Since the bus connector is attached to the middle of the line, both sides are available to supply charge and the effective charging impedance would be approximately 50 ohms. A 30-pF capacitance would have a charging time constant of 1.5 ns. This time constant fits the observations well during the rapid transient itself but does not fit the timing parameters of the voltage differences observed well after the rapid transient.

Elevated local temperatures are almost certainly present during the rapid transient (near the melting point of the metals!), so it seems plausible that the mystery voltage source is basically thermal electromotive force (EMF) between the pins. Allowing a few microseconds to achieve thermal equilibrium and subsequent loss of the thermal EMF also seems quite plausible. These details are inviting further detailed investigation but



(a) Demating Event with 60-microsecond Separation



(b) Demating Event with Bounce

Figure 22
Demating Events

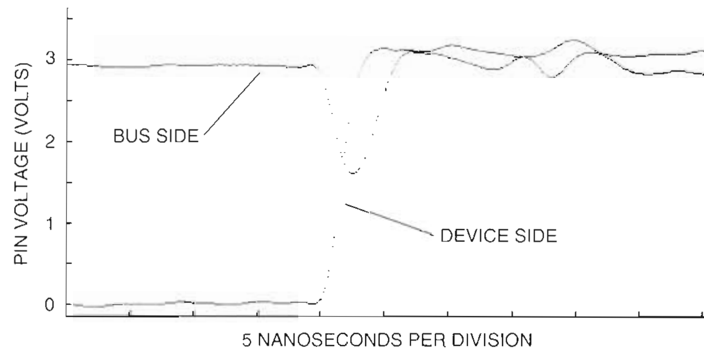


Figure 23
Most Severe Noise Pulse Observed

do not affect the practical conclusions as applied to parallel SCSI.

As added evidence for thermal effects, experiments with early LVD SCSI devices that use a 1.2-V bus level instead of the 3.5-V bus level shown in Figure 23 transfer much less energy and have a much shorter settling time before both sides of the contact track. These LVD results will be reported separately.

The point extracted from these charging-impedance and settling-time observations is simply that the overall energy transfer rate is limited by the microphysics of the process. This means that Figure 23 almost certainly illustrates the worst-case disturbances.

It has been noted that the bus pulse is similar to that produced by a stub on the bus and a signal with a fast rise/fall time. In a sense, we really are charging a stub in either case,

and in both cases the loaded or local characteristic impedance of the bus limits the extent of the disturbance.

To more accurately measure the noise pulse produced when a device is added to the bus, measurements were performed without a scope probe attached to the device pin. To do so required triggering the scope from the noise pulse on the bus side. Consequently, it was not possible to see the device-side charging dynamics. Figure 24 shows the measured pulse near the device connector and at a point 2 meters away.

The pulse measured in Figure 24 has approximately half the amplitude of the pulse in Figure 23. This is more reduction in amplitude than one would expect from the removal of 10 pF from the effective device capacitance, and this difference, while not completely explained, is in the favorable direction. The noise pulse that reached the next device (where it could be detected as an error) would be even smaller, because of the dispersion and attenuation in the bus and because the neighboring device would need to have its 25-pF capacitance charged also. The signal at the measurement point 2 meters away in Figure 24 indicates the intensity of the attenuation and dispersion to be expected in the rapid transient bus pulses. The details of the attenuation and dispersion depend somewhat on the bus media used.

The rapid transient bus pulses are shown on actual data pulses in Figure 25. The top trace in the figure shows a rapid transient pulse on a negated part of a single-ended SCSI signal. There is a scope probe on this device, but the device capacitance is only approximately 15 pF so the total with the probe is approximately 25 pF. Note that the noise pulse is approximately 0.8 V and does not take the signal into the receiver detection range below 2 V. This negated state is a bit higher than usually found, so the bus pulse is starting from a higher point. If the pulse had started from a lower point, for example about 2.5 V, the pulse amplitude would not have been as large. Further discussion of the receiver detection range appears later in this section.

The signals in Figure 25 were purposely chosen to have broad falling edges of approximately 15 ns. Normal SCSI signals are 5 ns or faster. The broad edges maximize the chance that the bus pulse will produce a signal slope reversal of the type that can produce multiple edges. The bottom trace in Figure 25 shows a bus pulse in the most sensitive part of the falling edge. This pulse produces almost no slope reversal because by the

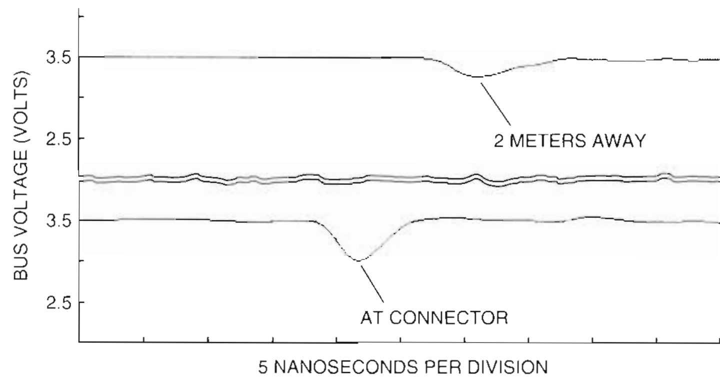


Figure 24
Bus Pulses with No Scope Probe Attached to the Device

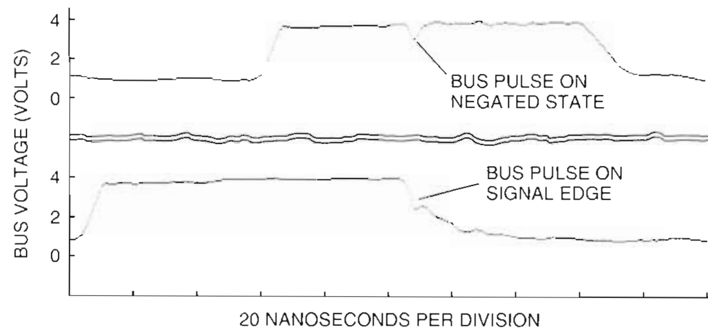


Figure 25
Bus Pulses on Actual Signals

time it is ready to become positive-going, the data signal has fallen so much that there is no voltage source to drive the signal more positive. At the beginning of the falling edge, the slew rate is increased by the bus pulse; in the middle, the edge is extended and consequently the overall time required for the falling edge is almost exactly the same as for the falling edge that has no bus pulse (see the top trace).

Therefore, the main effect of rapid transient pulses occurs when they intersect the signal edges (where a state change is expected anyway), and the effect is movement of the position of the edge by no more than 2 ns from the normal position. This movement is already accounted for in the SCSI standard as *pulse distortion skew*, so there is no important effect.

If the mating event happens while the bus signal is in the asserted state, there is little effect since little charge is transferred. If the event happens in the rising edge, there may not be enough voltage difference to start a rapid transient—again, there is little effect. If a rapid transient is initiated on a rising edge, the impact is still a small shift in the position of the edge. In any arbitrary combination of signal level and type of transient, the bus disturbance will not be greater than those shown in Figure 24 and Figure 25.

Differential

For differential SCSI systems, essentially the same behavior occurs as for the single-ended case except that the relationship between two contacts instead of just one must be considered. If insertion transients on the positive signal differential line are occurring at the same time as transients on the negative signal line, we must examine the difference between these transients to see what impact they have on the differential signals. Based on the time required between mating events on neighboring connector pins presented in the section Connector Insertion Dynamics and in Figure 15, it is evident that the differential case is almost always two independent and isolated single-ended cases. This is because the difference in the time required for different pins in the same connector to begin the mating process vastly exceeds the actual transient time on either signal.

In SCSI differential systems, both the positive and the negative signals are normally positive with respect to the local grounds. This means that the transients will be the same polarity on both signals.

In the very rare cases in which some overlap exists between the transient times on both signals, the rapid transient disturbances would usually be seen as common mode events that reduce the effective differential transient signal. These events are not seen if common mode noise exists where the signals have opposite polarity with respect to local grounds during the transients. In this case, it is theoretically possible to produce anticommon mode differential transients. However, the anticommon mode case will always have the positive and negative signal lines within a differential logical voltage level of ground, and the transients will therefore be small. Even in the anticommon mode case, the effect is at most a slight shift in the time when the differential state change is observed, since the transient disturbances are so small.

In the pathological differential case, large common mode levels exist on both the positive and the negative signals. The insertion transient will be larger because the bus voltage is larger. This case is even more rare since it requires both coincidental pin mating and coincidental large common mode.

The other case considered that can have a unique effect on differential systems is that of extended bounce. This case extends the effective mating time to the point when some overlap between the transient activity on the pins is more likely. Recall that the extended bounce case was only visible when a leakage mechanism was available to discharge the incoming device capacitance. In actual devices, no significant leakage occurs so a bounce event does not produce disturbances after this first contact.

The differential signal seen by the incoming device may be seriously affected by extended bounce if there is bus activity during this bounce. Consider, for example, a case in which the positive signal contact opened because of a bounce event after achieving a full charge. While it is still open, the negative signal changes state. Now both the positive and negative signals are at the same nominal potential, which is an indeterminate differential condition. Fortunately, this condition is not a problem because the only device that sees this condition is the device being inserted or removed and it is not in an operational state.

Summary of the Handling of Device Insertion and Removal Transients

After a complex, yet self-consistent, set of experimental data and interpretations, the concluding results are that the worst-case SCSI bus transients resulting from proper insertion and removal processes should not cause errors in the SCSI bus as presently specified in the SPI and Fast-20 (UltraSCSI) standards. The proper processes include pregrounding prior to insertion, avoiding excessive device capacitance, and using SCSI drivers and receivers that meet all of the SCSI requirements.[4]

As of this writing, all reports of device insertion/removal errors have been traced back to failure to use proper procedures or designs. The most common errors are lack of pregrounding, devices that do not maintain the high-impedance input state during power cycling, and power distribution or mechanical transient effects unrelated to SCSI proper.

The mechanisms that operate span a time spectrum from picoseconds in rapid transients to seconds in contact wipe and other macro connector operations.

The worst-case differential transients occur when one treats the differential system as two independent single-ended SCSI buses—one for the positive signal and one for the negative signal.

The rapid transient becomes more and more detectable as bus speeds increase and the receivers and timing margins become more sensitive. Schemes to encourage the gradual transient are the best protection against the ultimate problems caused by rapid transients. The best-known method for producing reliable gradual transients is to avoid a metal-to-metal contact during the initial contact and until the device capacitance is charged. At this time, no such connector system exists for SCSI applications.

Overall Summary

Evolution in four significant hardware technologies in the recent past has enabled parallel SCSI to break through the barriers that were preventing it from delivering excellent value, flexibility, and growth to the computer data storage industry. Application of more scientific methods, use of the latest silicon technology, and developments in the interconnect technology provided the foundation for these improvements. DIGITAL provided most of the basic data and led important standards and industry bodies to accomplish this.

Acknowledgments

Fee Lee, Keith Childs, Chuck Bagg, Pak Seto, and Jonathan Salles were instrumental in developing the special test environment and for acquiring much of the data presented in this paper. The author gratefully acknowledges the management and technical support from Pete Korce, Mike Chamberlain, Bob Passmore, Richie Lary, Ken Chester, Bob Rennick, Laura Woodburn, and Ellen Lary.

References

1. *ANSI X3.277-1996: Information Technology X3T9.2/375R SCSI-3 Fast-20, X3T10/1071D* (New York: American National Standards Institute, 1996). Available from Global Engineering, 15 Inverness Way East, Englewood, CO 80112-5704, tel. (800) 854-7159 or (303) 792-2181, fax (303) 792-2192.

2. Very High Density Cabled Interconnect (VHDCI) Specification, SFF-8441. Available from the SFF Committee, 14426 Black Walnut Court, Saratoga, CA 95070, voice fax-back service (408) 741-1600, tel. (408) 867-6630 ext. 303, fax (408) 867-2115.

3. *ANSI X.3.131-1994: Information Systems—Small Computer System Interface–2 (SCSI-2)*, X3T9.2/375R (New York: American National Standards Institute, 1994). Available from Global Engineering, 15 Inverness Way East, Englewood, CO 80112-5704, tel. (800) 854-7159 or (303) 792-2181, fax (303) 792-2192.

4. *ANSI X3.253-1995: Information Technology—SCSI-3 Parallel Interface (SPI), X3T10/855D* (New York: American National Standards Institute, 1995). Available from Global Engineering, 15 Inverness Way East, Englewood, CO 80112-5704, tel. (800) 854-7159 or (303) 792-2181, fax (303) 792-2192.

5. *SCSI Enhanced Parallel Interface (EPI), X3T10/1143D* (New York: American National Standards Institute, 1997). Available from Global Engineering, 15 Inverness Way East, Englewood, CO 80112-5704, tel. (800) 854-7159 or (303) 792-2181, fax (303) 792-2192.

6. Information about I/O interfaces is available at the T10 home page at http://www.symbios.com/t10.

7. Single Connector Attachment–2 (SCA-2) Specifications: SFF-8015, SFF-8046, SSF-8048, SSF-8066, and SSF-8451. Available from the SFF Committee, 14426 Black Walnut Court, Saratoga, CA 95070, voice fax-back service (408) 741-1600, tel. (408) 867-6630 ext. 303, fax (408) 867-2115.

Biography

William E. Ham
Bill Ham joined DIGITAL in 1983 and has been working in storage technology since 1990 as the manager of the Storage Bus Technical Office (SBTO). The SBTO group represents DIGITAL at most of the important standards and industry bodies that are involved with the transmission of storage data. Presently, these groups are SCSI, STA, Fibre Channel, and SFF. The SBTO group also creates information from actual laboratory testing and was responsible for introducing Fast-20 technology to ANSI in 1993 and hot-plugging technology to SCSI in 1992. Bill has a Ph.D. in electrical engineering from Southern Methodist University in Dallas, Texas, and has been active in the electronics industry in a variety of technologies since 1970. He has held significant positions in silicon chip, printed circuit board, multichip module technology, and storage bus technology. In addition, he is the past technical editor of the SSA physical standards, past editor of the new SPI-2 SCSI standard, editor of the entire family of SFF connector documents, editor of the new Enhanced Parallel Interface ANSI project for SCSI, and secretary for the Fibre Channel physical standards group (T11.2).

Peter L. Higginson
Michael C. Shand

Development of Router Clusters to Provide Fast Failover in IP Networks

IP networks do not normally provide fast failover mechanisms when IP routers fail or when links between hosts and routers break. In response to a customer request, a DIGITAL engineering team developed new protocols and mechanisms, as well as improvements to the DECNIS implementation, to provide a fast failover feature. The project achieved loss-of-service times below five seconds in response to any single failure while still allowing traffic to be shared between routers when there are no failures.

A DIGITAL router engineering team has refined and extended routing protocols to guarantee a five-second maximum loss-of-service time during a single failure in an Internet Protocol (IP) network. We use the term *router cluster* to describe our improved implementation. A router cluster is defined as a group of routers on the same local area network (LAN), providing mutual backup. Router clusters have been in service since mid-1995.

Background

The Digital Equipment Corporation Network Integration Server (DECNIS) bridge/router is a midrange to high-end product designed and built by a DIGITAL Networks Product Business Group in Reading, U.K.[1] The DECNIS performs high-speed routing of IP, DECnet, and OSI (open system interconnection) protocols and can have the following network interfaces: Ethernet, FDDI (fiber distributed data interface), ATM (asynchronous transfer mode), HSSI (High-Speed Serial Interface), T1/E1 (digital transmission schemes), and lower-speed WAN (wide area network) interfaces. The DECNIS bridge/router is designed around a Futurebus backplane, with a number of semi-autonomous line cards, a hardware based address lookup engine, and a central control processor responsible for the control protocols and route calculation. Data packets are normally handled completely by the line cards and go to the central processor only in exception cases.

The DECNIS routers run a number of high-profile, high-availability, wide-area data networks for telephone service providers, stock exchanges, and chemical companies, as well as forming the backbone of DIGITAL's internal network.

Typically, the DECNIS routers are deployed in redundant groups with diverse interconnections, to provide very high availability. A common requirement is never to take the network down (i.e., during maintenance periods, connectivity is preserved but redundancy is reduced).

Overview

IP is the most widely used protocol for communication between hosts. Routers (or gateways) are used to link hosts that are not directly connected. When IP was originally designed, duplication of WAN links was common but duplication of gateways for hosts was rare, and no mechanisms for avoiding failed routers or broken links between hosts and routers were developed.

In 1994, we began a project to restrict loss-of-service times to below five seconds in response to any single failure; for example, failure of a router or its electrical supply, failure of a link between routers, or failure of the connection between the router and the LAN on which the host resides. In contrast, existing routing protocols have recovery times in the 30- to 45-second range, and bridging protocols are no better. Providing fast failover in IP networks required enhancements to many areas of the router's design to cover all the possible failure cases. It also required the invention of new protocols to support the host-router interaction under IP. This was achieved without requiring any changes to the host IP code.

In this paper, we start by discussing our targets and the behavior of existing routing or bridging protocols and follow this with a detailed analysis of the different failure cases. We then show how we have modified the behavior of the routing control protocols to achieve the desired failover times on links between routers or in response to the failure of intermediate routers. Finally, we describe the new IP Standby Protocol and the mechanisms we developed to achieve fast recovery from failures on the LANs local to the end hosts. This part of the problem is the most challenging because the hosts are of many types and have IP implementations that cannot realistically be changed. Thus all changes have to be made in the routers.

Our secondary aims were to allow the use of router clusters in any existing network configuration, not to constrain failover to simple pairs of routers, to be able to share traffic between available routers, and to continue to use the Internet Control Message Protocol (ICMP) redirect mechanism for optimum choice of router by hosts on a per destination basis. A common problem of hosts is that they do not time out redirects. This problem is avoided by the adoption mechanism within the router cluster. Having met these aims, as well as fast failover, we can justifiably call the result router clusters.

The Customer Challenge

A particular customer, a telecommunications service provider, has an Intelligent Services Network application by which voice calls can be transferred to another operator at a different location. The data network manages the transferral and passes information about the call. The application uses User Datagram Protocol (UDP) packets in IP with retransmission from the application itself.

Because this application requires a high level of data network availability, network designers planned a duplicate network with many paired links and some mesh connections. Particular problems arise when the human initiator becomes impatient if there are delays; however, the more critical requirement was one over which the network designers had no control. The source of the calls is another system that makes a single high-level retransmission after five seconds. If that retransmission does not receive a response, the whole system at the site is assumed to have failed. This leads to new calls being routed to other service sites or suppliers, and manual intervention is required.

To resolve this issue, the customer requested a networking system that would recover from a single failure in any link, interface, or router within a five-second period. The standard test (which both the customer and we use) is to start a once-per-second ping, and to expect to drop no more than four consecutive ping packets (or their responses) upon any event. The five-second maximum break also has to apply to any disruption when the failed component recovers.

To meet the customer challenge, the router group in Reading developed the router cluster implementation on the DECNIS. In the next two sections, we discuss the bridging and routing protocols in use at the start of our project and relate our analysis of the customer's network problems.

Bridging and Routing Default Recovery Times

In a large network, a routing control protocol is essential in order to dynamically determine the topology of the network and to detect failing links. Bridging control protocols may be used similarly in smaller networks or may be used in combination with routing.

Bridging and routing control protocols often have failure recovery times in the order of a minute or more. A typical recovery consists of a detect time during which adjacent routers learn about the failure; a distribution time during which the knowledge is shared, possibly throughout the whole network; and a route recalculation time during which a new set of routes is calculated and passed to the forwarding engine.

Detection times are in the order of tens of seconds; for example, 30 seconds is a common default. The two most popular link-state routing control protocols in large IP networks are Open Shortest Path First (OSPF)[2] and Integrated Intermediate System-to-Intermediate System (Integrated IS-IS).[3] These protocols have distribution "hold downs" (to limit the impact of route flaps) to prevent the generation of a

new control message within some interval (typically 5 or 30 seconds) of a previous one. The distribution of the new information is rapid (typically less than one second), depending primarily on link speeds and network diameter; however, the distribution may be adversely affected by transmission errors which require retransmission. The default retransmission times after packet loss vary between 2 and 10 seconds. The route recalculation typically takes less than one second. These values result in total recovery times after failures (for routing protocols with default settings) in the 45- to 90-second range.

Distance vector routing protocols, such as the Routing Information Protocol (RIP),[4] typically take even longer to recover, partly because the route computation process is inherently distributed and requires multiple protocol exchanges to reach convergence, and partly because their timer settings tend to be fixed at relatively long settings. Consequently, their use is not further considered in this paper.

Similarly, bridging protocols, as standard, use a 15-second timer; one of the worst-case recovery situations requires three timeouts, making 45 seconds in all. Another bridging recovery case requires an unsolicited data packet from a host and this results in an indeterminate time, although a timeout will cause flooding after a period.

In IP protocols, there is no simple way for a host to detect the failure of its gateway; nor is it simple for a router to detect the failure to communicate with a host. In the former case, several minutes may pass before an Address Resolution Protocol (ARP) entry times out and an alternative gateway is chosen; for some implementations, recovery may be impossible without manual intervention. Failure to communicate with a host may be the result of failure of the host itself, which is outside the scope of this project. Alternatively, it may be due to failure of the LAN, or the router's LAN interface. In this case, there exists an alternative route to the LAN through another router, but the routing protocols will not make use of it unless the subnet(s) on the LAN are declared unreachable. This requires either manual intervention or timely detection of the LAN failure by the router.

Analysis of the Failure Cases

The first task in meeting the customer's challenge was to analyze the various failure and recovery modes and determine which existing management parameters could be tuned to improve recovery times. After that, new protocols and mechanisms could be designed to fill the remaining shortcomings.

A typical network configuration is shown in Figure 1. The target network is similar but has more sites and many more hosts on each LAN. Many of the site

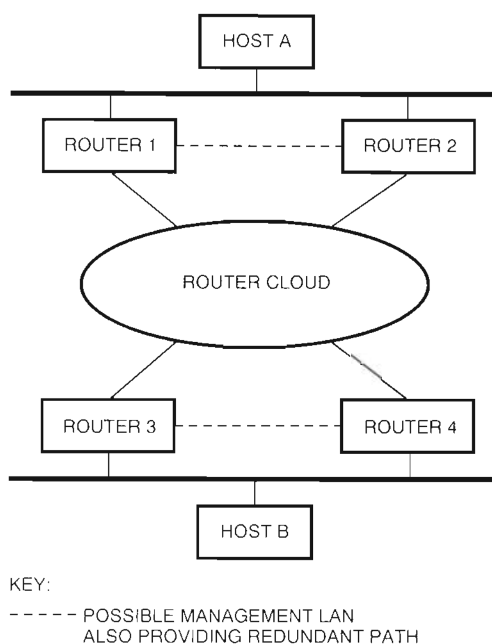


Figure 1
Typical Configuration for Router Cluster Use

routers are DECNIS 500 routers with one or two WAN links and two Ethernets. The second Ethernet is used as a management rail and as a redundant local path between routers one and two (R1–R2) and between routers three and four (R3–R4).

In the original plans for the customer network, the router cloud consisted of groups of routers at two or three central sites and pairs of links to the host sites. In designing our solution, however, we tried to allow any number of routers on each LAN, interconnected by a general mesh network. For test purposes, both we and the customer used this set-up with direct R1–R3 and R2–R4 T1 links as the network cloud.

We have to consider what happens to packets traveling in each direction during a failure: there is little gain in delivering the data and losing the acknowledgments. Since the direction of data flow does not give rise to additional complications in the network cloud, there are just two failure cases:

1. Failure of a router in the network cloud
2. Failure of a link in the network cloud

We keep these cases distinct because the failure and recovery mechanisms are slightly different.

We also need to consider a failure local to one of the LANs on which the hosts are attached. A failure here has two consequences: (1) The packets originated by the host must be sent to a different router, and (2) The response packets from the other host through the network cloud must also be sent to a different router, so

that it can send them to the host. We break down this type of failure into the following three cases:

3. Packets from the host to a failed or disconnected router
4. Packets to the host when the router fails
5. Packets to the host when the router interface fails

Note that we are using the term *router interface failure* to include cases in which the connector falls out or some failure occurs in the LAN local to the router (such that the router can detect it). In practice, failure of an interface is rare. (Removing the plug is not particularly common in real networks but is easy to test.) Figure 2 shows these failure cases; this configuration was also used for some of the testing.

Recovery of a link that previously failed causes no problems because the routers will not attempt to use it until after it has been detected as being available. Prior to that, they have alternate paths available. Recovery of a failed router can cause problems because the router may receive traffic before it has acquired sufficient network topology to forward the traffic correctly. Recovery of a router is discussed more fully in the section on Interface Delay.

Can Existing Bridging or Routing Protocols Achieve 5-Second Failover in a Network Cloud?

In this section, we discuss the failure of a router and the failure of a link in the network cloud (cases 1 and 2). The customer requested enhanced routing, and the existing network was a large routed WAN, so enhancing bridging was never seriously considered. Our experience has shown that the 15-second bridge timers can be reduced only in small, tightly controlled networks and not in large WANs. Consequently, bridging is unsuitable for fast failover in large networks.

For link-state routing control protocols such as OSPF and Integrated IS-IS, once a failure has been detected recovery takes place in two overlapping phases: a flood phase in which information about the failure is distributed to all routers, and a route calculation phase in which each router works out the new routes. The protocols have been designed so that only local failures have to be detected and manageable parameters control the speed of detection.

Detection of failure is achieved by exchanging Hello messages on a regular basis with neighboring routers. Since the connections are usually LAN or Point-to-Point Protocol (PPP) (i.e., with no link-layer acknowledgments), a number of messages must be missed before the adjacency to the neighbor is lost. The messages used to maintain the adjacency are independent of other traffic (and in a design like the DECNIS may be the only traffic that the control processor sees). Typical default values are messages at three-second intervals and 10 lost for a failure, but it is possible to reduce these.

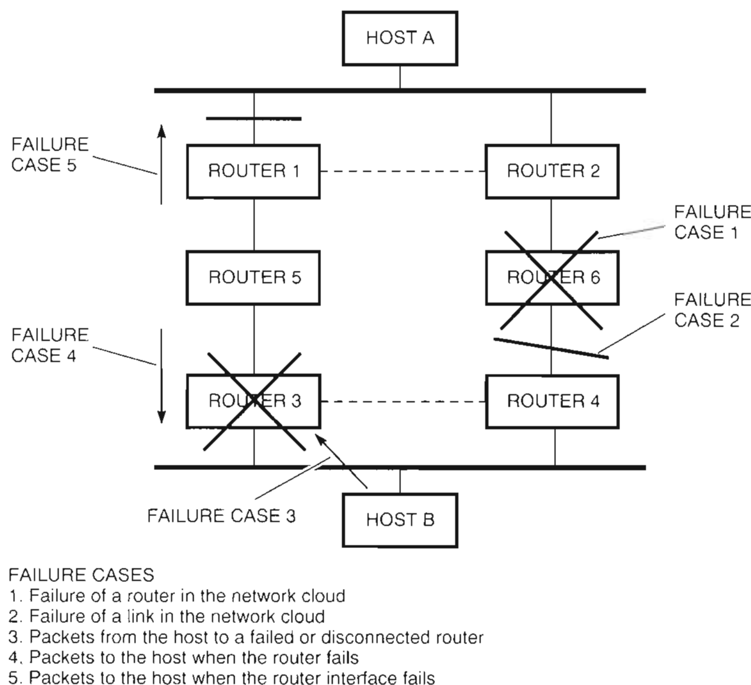


Figure 2
Diagram of Failure Cases Targeted for Recovery

Decreasing the Routing Timers

The default timer values are chosen to reduce overheads, to cover short outages, and to ensure that it is not possible for long packets to cause the adjacency to expire accidentally by blocking Hello transmission. (Note transmission of a 4,500-byte packet on a 64 kilobit-per-second link takes half a second, and queuing would normally require more than a packet time.) However, with high-quality T1 or higher link speeds in the target network and priority queuing of Hellos in the DECNIS, it is acceptable to send the Hellos at one-second intervals and count three missed as a failure. (Although we have successfully tested counts of two, we do not recommend that value for customers on WAN links because a single link error combined with a delay due to a long data packet would cause a spurious failure to be detected.) The settings of one second and three repeats were within the existing permitted ranges for the routing protocols.

When these shorter timers are used, it is important that any LANs in the network should not be overloaded to the extent that transmissions are delayed. The network managers should monitor WAN links and disable any links that have high error rates. Given the duplication of routes, it is better to disable and initiate repairs to a bad link than to continue a poor service. Many customers, with less controlled networks and less aggressive recovery targets, have adopted the router cluster system but kept to more conservative timers (such as 1 second and 10 repeats).

Implementation and Testing Issues

In some cases, a failed link may be detected at a lower level (e.g., modem signals or FDDI station management) well before the routing protocol realizes that it has stopped getting Hellos and declares the adjacency lost. (This can lead to good results during testing, but it is essential also to test link-failure modes that are not detected by lower levels.) In the worst case, however, both the detection of a failed router or the detection of a failed link rely on the adjacency loss and so have the same timings.

Loss of an adjacency causes a router to issue a revised (set of) link-state messages reflecting its new view of the local topology. These link-state messages are flooded throughout the network and cause every router in the network to recalculate its route tables. However, because the two or more routers will normally time out the adjacency at different times, one message arrives first and causes a premature recalculation of the tables. Therefore it may require a subsequent recalculation of the route tables before a new two-way path can be utilized. We had to tune the router implementation to make sure that subsequent recalculations were done in a speedy manner.

During initial testing of these parameters, we discovered that failure of certain routers represented a more serious case. However discussion of this is deferred to the later section The Designated Router Problem.

Our target five seconds is made up of three seconds for the failure to be detected, leaving two seconds for the information about the failure to be flooded to all routers and for them to recalculate their routes. Within the segment of the network where the recovery is required, this has been achieved (with some tuning of the software).

Recovery from Failures on the LANs Local to the End Hosts

The previous section shows that we can deal with router failure and link failure in the network cloud (cases 1 and 2). Here we consider cases 3, 4, and 5, those that deal with failures on the LANs local to the end hosts.

From the point of view of other routers, a failed router on a LAN (case 4) is identical to a failed router in the network cloud (case 1): a router has died, and the other routers need to route around it. Failure case 4 therefore is remedied by the timer adjustments described in the previous section. Note that these timer adjustments are an integral part of the LAN solution, because they allow the returning traffic to be re-routed. These timer adjustments cannot work properly if the LAN parts of router clusters are using an inappropriate routing control protocol such as RIP[4], which takes up to 90 seconds to recover from failures.

Detecting LAN Failure at the Router

A solution to case 5—packets to the host when the router interface fails—for IP requires that the router can detect a failure of its interface (for example, that the plug has been removed). If the LAN is an FDDI, this is trivial and virtually instantaneous because continuous signals on the ring indicate that it is working and the interface directly signals failure. For Ethernet, we faced a number of problems, partly due to our implementation and partly due to the nature of Ethernet itself. We formed a small team to work on this problem alone.

Because of the variety of Ethernet interfaces that might be attached, there is no direct indication of failure: only an indirect one by failure to successfully transmit a packet within a one-second interval. For maximum speed, the DECNIS implementation queues a ring of eight buffers on the transmit interface and does not check for errors until a ring slot is about to be reused. This means that an error is only detected some time after it has occurred, consuming much of our five-second budget.

The control software in the DECNIS management processor has no direct knowledge of data traffic because it passes directly between the line cards. Therefore it sends test packets at regular intervals to find out if the interface has failed. By sending large test packets occupying many buffers, it ensures that the ring circulates and errors are detected. Initially, we

reduced the timers and increased the frequency of test packets to be able to detect interface failure within three seconds. (The test packets have the sender as destination so that no one receives them and, as usual, more than one failure to transmit is required before the interface is declared unusable.)

This initial solution caused several problems when it was deployed to a wider customer group; we had more complaints than previously about the bandwidth consumed by the test messages and, more seriously, a number of instances of previously working networks being reported as unusable. These problem networks were either exceptionally busy or had some otherwise undetected hardware problem. Over time, the networks with hardware problems were fixed, and we modified the timers to avoid false triggering on very busy networks. Clearly, the three-second target required more thought.

Several enhancements have since been made. First, the timers are user configurable so that the network managers can trade off between aggressive recovery times, bandwidth used, and false detection. Second, the test packet generator takes into account other packets sent by the control processor such that they are only sent to the size and extent required for the total traffic to cause the ring to circulate. This is a significant improvement because the aggressive routing timers discussed previously cause Hello packets to be sent at one-second intervals, which is often sufficient not to require extra test packets. Third, the line card provides extra feedback to the control program about packets received and the transmission of packets not originated by the control processor. This feedback gives an indication of successful operation even if some transmits are failing.

Re-routing Host Traffic When a Router or Router Connection Fails

Case 3 was by far the most difficult problem to solve. IP does not provide a standard mechanism to re-route host traffic when a router fails, and the only method in common use (snooping RIP messages in the hosts) is both "deprecated" by the RFCs and has fixed 45-second timers that exceed our recovery target. Customers have a wide range of IP implementations on their hosts, and reliance on nonstandard features is difficult. The particular target application for this work ran on personal computer systems with a third-party IP stack, and we obtained a copy for testing. Such IP stacks sometimes do not have sophisticated recovery schemes and discussion with various experts led us to believe that we should not rely on any co-operation from the hosts.

Among other objectives, we wanted to be independent of the routing control protocol in use (if any), to permit both a mesh style of networking and more than two routers in a cluster, and to continue to route traffic by reasonably optimal routes. In addition, we wished to not confuse network management protocols about the true identity of the routers involved and, if possible, to share traffic over the WAN links where appropriate.

Electing a Primary Router

In our solution, the first requirement is for other routers on the LAN to detect that a router has failed or become disconnected, and to have a primary router elected to organize recovery. This is achieved by all routers broadcasting packets (called IP Standby Hellos) to other routers on the LAN every second. The highest priority (with the highest IP address breaking ties) router becomes the primary router, and failure to receive IP Standby Hellos from another router for *n* seconds (three is the default) causes it to be regarded as disconnected. This condition may cause the selection of a new primary router, which would then initiate recovery to take traffic on behalf of the disconnected router.

The IP Standby Hellos are sent as "all routers multicasts" and therefore do not add additional load to hosts. They are UDP datagrams[5] to a port we registered for this purpose (digital-vrc; see the Internet Assigned Numbers Authority [IANA] on-line list). The routers are manually configured with a list of all routers in the cluster. To make configuration easier and less error prone, the list on each router includes itself, and hence an identical set of configuration parameters can be used for all the routers in a cluster. Automatic configuration was rejected because of the problem of knowing which other routers should exist.

Function of the Primary Router in ARP Mode

Our first attempt (called ARP Mode) uses a fake IP address (one per subnet for a LAN with multiple subnets), which the current primary router adopts and the hosts have configured as their default router. The primary router returns its own media access control (MAC) address when the host broadcasts an ARP request (using the standard ARP protocol[6]) for the fake IP address and thus takes the traffic from the host. After a failure, a newly elected primary router broadcasts an ARP request containing the information that the fake IP address is now associated with the new primary router's MAC address. This causes the host to update its tables and to forward all traffic to the new primary router.

The sending of ICMP redirects[7] by the routers has to be disabled in ARP mode. Redirects sent by a router would cause hosts to send traffic to an IP address other than the fake IP address controlled by the cluster, and recovery from failure of that router would then be impossible. Disabling redirects causes an additional

problem. If the primary router's WAN link fails, all the packets have to be inefficiently forwarded back over the LAN to other routers. To avoid this problem, we introduced the concept of monitored circuits, whereby the priority of a router to become the primary depends on the state of the WAN link. Thus, the primary router changes when the WAN link fails (or all the links fail if there are several), and the hosts send the packets to the new primary (whose WAN link is still intact).

ARP mode has a number of disadvantages. It does not necessarily use an optimum route when the WAN links form a mesh rather than the simple pair case, because redirects have to be disabled. The monitored circuit concept works only on the first hop from the router; more distant failures cannot change the IP Standby priority and may result in inefficient routing. Most seriously, the rules for hosts acting on information in ARP requests have only a "suggested implementation" status in the RFCs, and we found several hosts that did not change when requested or were very slow in doing so. (Note that we did consider broadcasting an ARP response, but there is no allowance in the specifications for this message to be a broadcast packet, whereas an ARP request is normally a broadcast packet.)

MAC Mode IP Standby (to Re-route Host Traffic)

To solve these problems, we looked for a mechanism that did not rely on any host participation. The result was what we termed MAC mode. Here, each router uses its own IP address (or addresses for multiple subnets) but answers ARP requests with one of a group of special MAC addresses, configured for each router as part of the router cluster configuration. When a router fails or becomes disconnected, the primary (or the newly elected primary) router adopts the failed router. By adopt, we mean it responds to ARP requests for the failed router's IP address with the failed router's special MAC address, and it receives and forwards all packets sent to the failed router's special MAC address (in addition to traffic sent to the primary router's own special MAC address and those of any other failed routers it has adopted).

The immediate advantages of MAC mode are that ICMP redirects can continue to be used, and, providing the redirects are to routers in the cluster, the fast failover will continue to protect against further failures. The mechanism is completely transparent to the host. In a cluster with more than two routers, the primary router will use redirects to cause traffic (resulting from failure) to use other routers in the cluster if they have better routes to specific destinations. Thus multiple routers in a cluster and mesh networks are supported. This also solves the problem of hosts not timing out redirects (an omission common to many IP implementations derived from BSD), because the redirected address has been adopted.

In MAC mode, the hosts are configured with the IP address of any router in the cluster as the default gateway. (The concept that it does not matter which router is chosen is one of the hardest for users to accept.) Some load sharing can be achieved by setting different addresses in different hosts.

Since the DECNIS is a bridge router, it has the capability to receive all packets on Ethernet and many MAC addresses on FDDI; thus all packets on all the special MAC addresses are seen by all routers in the cluster, and its own and those of any adopted routers are forwarded. The special MAC addresses used are those associated with the unused DECnet area 0. They are ideal because they are part of the locally administered group and have implementation efficiencies in the DECNIS because the DECnet hi-ord (AA-00-04-00) is already decoded, and they are 16 addresses differing in one nibble only (i.e., AA-00-04-00-0x-00, where x is the hexadecimal index of the router in the cluster). Note that ARP requests sent by the router must also contain the special MAC address in the source hardware address field of the ARP packet, otherwise the hosts' ARP tables may be updated to contain the wrong MAC address.

MAC mode has minor disadvantages. Initially, it is easy to spread the load over a number of routers; however, this can be lost after redirects. In addition, a small chance of packet duplication exists during recovery because there may be a short period when both routers are receiving on the same special MAC address (which does not happen in ARP mode because the host changes the MAC address it is using). This is preferable to a period when no router is receiving on that address.

Interface Delay

Recently, we added an interface delay option to ameliorate a situation more likely to occur in large networks. In this situation, a router, rebooting after a power loss, a reboot, or a crash, reacquires its special MAC address before it has received all of the routing updates from neighboring routers and thus drops packets sent to it (and worse, returns "unreachable" to the host). Typically, the main LAN initialization would be delayed for 30 seconds while routing table updates were received over the WAN interfaces and any other LAN interfaces. The backup continues to operate during this 30 seconds. (Note that with Integrated IS-IS, we could have delayed IP on the whole router, but we did not do this because it would not have worked for OSPF, which requires IP to do the updates.) We use a fixed configurable time rather than attempting to detect the end of updating, because determining completion is difficult if the network is in a state of flux or the router's WAN links are down.

Redirects and Hosts That Ignore Them

When a router issues an ICMP redirect, the RFCs state that it must include its own IP address in the redirect packet. A host is required to ignore a redirect received from a router whose IP address is not the host's next hop address for the particular destination address. Therefore, it is necessary to ensure that the address of the failed router is correctly included when issuing a redirect on its behalf. In the DECNIS implementation, because the destination MAC address of a received packet is not available to the control processor, the primary router cannot tell whether a redirect has to be issued on behalf of itself or one of the adopted routers. The primary router therefore issues multiple redirects—one for each adopted router (in addition to its own). Since redirects are rare, this is not a problem, but they could be avoided by passing the MAC destination address of the original packet (or just five bits to flag a special MAC address and say which it is) to the control processor.

It is contrary to the basic IP rules for hosts to ignore redirects.[8] Despite the rules, some hosts do ignore redirects and continue sending traffic which has to be sent back over the same LAN. These cause problems in all networks because of the load, and, in the DECNIS implementation, because every time the line card recognizes a redirect opportunity, it signals the control processor to consider sending a redirect. This may happen at data packet rates and is a severe load on the control processor, which slows down processing of routing updates and might then cause our five-second recovery target to be exceeded.

To reduce the problems caused by hosts ignoring redirects, we improved the implementation to rate-limit the generation of redirect opportunity messages by the line cards. We also recommend in the documentation that, where it is known that hosts ignore redirects (or their generation is not desired), the routers be connected by a lower-cost LAN than the main service LAN (such as the management LANs shown in Figure 1). Normally, this would mean linking (just) the routers by a second Ethernet and setting its routing metric so that it is preferred to the main LAN for packets that would otherwise traverse back on the main LAN to the other router. This has two advantages. Such packets do not consume double bandwidth and cause congestion on the main LAN, and they pass only through the fast-path parts of the router, which are well able to handle full Ethernet bandwidth.

In MAC mode, it is also possible to define a router that does not actually exist (but has an IP address and a special MAC address) and is adopted by another router, depending on the state of monitored WAN circuits. Setting this as the default gateway is another way of coping with hosts that ignore redirects.

Special Considerations for Bridges

We do not recommend putting a bridge or layer 2 switch between members of a router cluster, because during failover, action would be required from the bridge in order for the primary router to receive packets that previously were not present on its side of the bridge. We cannot rely on this being the case, so we must have a way of allowing bridges to learn where the special MAC addresses currently are. More importantly, if bridges do not know where the special MAC addresses are, they often use much less efficient (flooding) mechanisms.

For greater traceability (and simpler implementation), we use the router's real MAC address as the source address in data packets that it sources or forwards. We use the special MAC address as the source address in the IP Standby Hellos. Since the Hello is sent out as an IP multicast, it is seen by all bridges or switches in the local bridged network and causes them to learn the location of the address (whereas data packets might not be seen by non-local bridges). Since we are sending the Hellos every one second anyway, there is no extra overhead.

When a primary router has adopted routers, it cycles the source MAC address used for sending its Hello between its own special MAC address and those of the adopted routers. We also send out an additional Hello immediately when we adopt a router to speed up recognition of the change.

Since the same set of special MAC addresses is used by all router clusters, we were concerned that a bridge that was set up to bridge a non-IP protocol (e.g., local area transport [LAT]) but not to bridge IP, might be confused to see the same special MAC address on more than one port. (This has been observed to happen accidentally, and the resultant meltdown has led us to avoid any risk, however slight, of this happening.) Hence we make 16 special MAC addresses available and recommend to users that they allocate them uniquely within a bridged domain, or at least use disjoint sets on either side of a bridge.

The Designated Router Problem

While testing router failures, we discovered additional delays during recovery due to the way in which link-state protocols operated on LANs. In these cases, the failure of routers not handling the data packets can also result in interruption of service due to the control mechanisms used.

For efficiency reasons in link-state routing protocols, when several routers are connected to a LAN, they elect a designated router and the routing protocols treat the LAN as having a single point-to-point connection between each real router and a pseudo router maintained by the designated router (rather

than connections between all the routers). The designated router issues link-state packets on behalf of the pseudo router, showing it as having connections to each real router on the local LAN, and each router issues a link-state packet showing connection to the pseudo router. This mechanism operates in a broadly similar way in both Integrated IS-IS and OSPF; the primary difference being that the OSPF election exhibits hysteresis, thus minimizing unnecessary designated router changes.

For routing table calculations, a transit path over the LAN is taken from a router to the pseudo router and then to another router on the LAN. Hence any change in pseudo router status disrupts calculation of the network map.

When a designated router fails, a slew of updates occurs; each router on the LAN loses the adjacency to the old designated router and issues a new link-state packet. Next, the new designated router is elected (or in the case of OSPF, the backup designated router takes over), and each router issues a link-state packet showing a link to it. In parallel, the new designated router issues a set of link-state packets showing its connections. This is a new router on the network as far as the other routers are concerned; the old designated router stays, disconnected, in the tables for as long as 20 minutes to an hour. This happens at level 1 and at level 2 in Integrated IS-IS, resulting in twice as many updates. The interactions are complex; in general, they result in the sending of multiple, new link-state messages.

Apart from the pure distribution and processing problem of these updates and new link-state packets, there are deliberate delays added. A minor one is that updates in Integrated IS-IS are rate-limited on LANs (to minimize the possibility of message loss). A major one is that a particular link-state packet cannot be updated within a holding time from a previous update (to limit the number of messages actually generated). The default holding time is 30 seconds in Integrated IS-IS; it can be reduced to 1 second in the event we found that the best solution was to allow as many as 10 updates in a 10-second period. The reason for this is that the first update usually contains information about the disconnection and it is highly desirable to get the update with the connection out as fast as possible. In addition, in the wider network, an update can overtake and replace a previous one.

With OSPF, the protocol defines a minimum holding time of five seconds, which limits the recovery time when the designated router fails. The target customer's network was using Integrated IS-IS, and so we were able to achieve the five-second recovery even when the designated router failed. (Note that with two routers, one must be the designated router so it is not a rare case.) We have not, so far, felt that it is worthwhile to break the rules by allowing a shorter holding time for OSPF.

Conclusions

We successfully designed and implemented router clusters for the DECNIS router with shared workload and interruptions after failures of less than five seconds in both LAN and WAN environments. This capability has been deployed in the product since the middle of 1995. An Internet Engineering Task Force (IETF) group is currently attempting to produce a standard protocol to meet this need.[9]

Acknowledgments

Various members of the router engineering team in Reading, U.K. assisted with ideas for this work. In particular, we must mention Dave Forster who implemented the high-level IP changes, Chris Szmidt who implemented the line card forwarding, and John Rigby who implemented the bit in-between and the Ethernet cable-out detection.

References

1. D. Brash and S. Bryant, "The DECNIS 500/600 Multiprotocol Bridge Router and Gateway," *Digital Technical Journal*, vol. 5, no. 1 (Winter 1993): 84–98.

2. J. Moy, "OSPF Version 2," Internet Engineering Task Force, RFC 1583 (March 1994).

3. R. Callon, "Use of OSI IS-IS for Routing in TCP/IP and Dual Environments," Internet Engineering Task Force, RFC 1195 (December 1990).

4. C. Hedrick, "Routing Information Protocol," Internet Engineering Task Force, RFC 1058 (June 1988).

5. J. Postel, "User Datagram Protocol," SRI Network Information Center, Menlo Park, Calif., RFC 768 (August 1980).

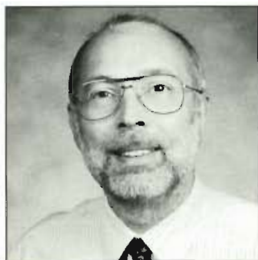
6. D. Plummer, "Ethernet Address Resolution Protocol," Internet Engineering Task Force, RFC 826 (November 1982).

7. J. Postel, "Internet Control Message Protocol," Internet Engineering Task Force, RFC 792 (September 1981).

8. R. Braden, "Requirements for Internet Hosts—Communication Layers," Network Information Center, RFC 1122 (October 1989).

9. R. Hinden, S. Knight, D. Weaver, D. Whipple, D. Mitzel, P. Hunt, P. Higginson, and M. Shand, "Virtual Router Redundancy Protocol," Internet Drafts <draft-ietf-vrrp-spec-03.txt> (October 1997).

Biographies



Peter L. Higginson
Peter Higginson manages the advanced development work on router products for DIGITAL's Internetworking Products Engineering Group in Reading U.K. (IPEG-Europe). His responsibilities include improving communications on customers' large networks. Most recently, he contributed to the corporate Web gateway strategy and future router products. Peter was issued one patent on efficient ATM cell synchronization and has applied for several other patents related to networks. He has published many papers, including one on a PDP-9 for DECUS in 1971. Before joining DIGITAL in 1990, Peter was the software director for UltraNet Ltd. (now part of the Anite Group), a maker of X.25 equipment. For 12 years before that, he was a lecturer in the Department of Computer Science, University College London. He received an M. Sc. in computer science from University of London in 1970 and a B. Sc. (honours) in mathematics from University College London in 1969. Peter connected the first non-U.S. host to the Arpanet in 1973.

Michael C. Shand
Mike Shand is a consulting software engineer with DIGITAL's Network Products Business in Reading, U.K. He is currently involved in the design of IP routing algorithms and the system-level design of networking products. Formerly, Mike was a member of the NAC (Networks and Communications) Architecture Group where he designed DECnet OSI, Phase V routing architecture. Before joining DIGITAL in 1985, Mike was the assistant director (systems) of the Computing Centre at Kingston University. He earned an M.A. in the natural sciences from the University of Cambridge in 1971 and a Ph. D. in surface chemistry from Kingston University in 1974. He was awarded six patents (and has filed another) in various aspects of networking.

Lawrence G. Palmer
Ricky S. Palmer

Shared Desktop: A Collaborative Tool for Sharing 3-D Applications among Different Window Systems

The DIGITAL Workstations Group has designed a software application that supports the sharing of three-dimensional graphics and audio across the network. The Shared Desktop application enables the collaborative use of any application over local and long-distance networks, as well as interoperation among Windows- and UNIX-based computers. Its simple user interface employs screen capture and data compression techniques and a high-level protocol to transmit packets using TCP/IP over the Internet.

An advanced product development effort undertaken by graphics engineers in the DIGITAL Workstations Group led to the creation of a new software application called Shared Desktop. One project goal was to enable collaboration among users of three-dimensional (3-D) graphics workstations that run either the UNIX or the Windows NT operating system. Another goal was to allow these users to access the high-performance 3-D capabilities of their office workstations from their laptop computers or home-based personal computers (PCs) that run the Windows 95 system and do not have 3-D graphics hardware. This goal necessitated a cross–operating-system application that could efficiently and effectively handle 3-D graphics in real time and share these graphics with machines such as laptop computers and PCs.

In this paper, we begin with a discussion of the software currently available for computer collaboration. We then discuss the development of the Shared Desktop application, focusing on the user interface, protocol splitting, screen capture and data handling, and dissimilar frame buffers. We conclude with sections on additional uses and future directions of the Shared Desktop product.

Current Collaboration Software

Computer collaboration may be defined as the interaction between computers and their human users over a local or long-distance network. In general, it involves a transfer of textual, graphical, audible, and visual information from one collaborator to another. The participants share control information either by means of computer generated synchronization events or by human voice and visual movement.[1]

Specifically, computer collaboration involves communicating and sharing data among participants who can be located anywhere in a building, a city, a country, or the world. Each participant has either a PC, a workstation, or a laptop computer. Some machines contain 3-D graphics adapters with hardware acceleration. (Computer-aided design/computer-assisted manufacturing [CAD/CAM] applications like Parametric Technology Corporation's [PTC] Pro/ENGINEER use hardware accelerators through OpenGL[2] or

Direct3D[5] programming protocols.) Other computers do not contain 3-D accelerator boards and provide 3-D capabilities through software-only routines on two-dimensional (2-D) hardware. In a typical collaboration, a person wanting to share a specific 3-D graphical display of a part or model telephones others to discuss the design in progress. After the initial contact, the collaborators may continue the telephone call or switch to the audio function of the application. The graphics part appears on each participant's screen along with associated keyboard and mouse events. As the collaborators discuss the work, they may each interact with the display to highlight, rotate, and change the look or design of the 3-D part. In this way, even though the participants are separated by some distance, they may interact as if they were all sitting around a table working, conversing, and designing the 3-D part.

Current software that facilitates computer-based collaboration runs through a range of capabilities from the earliest forms of electronic mail to the most recent offerings of complete collaborative sharing of the computer. Examples include WinFrame technology from Citrix Systems, Inc., NetMeeting from Microsoft Corporation, Netscape Communicator from Netscape Communications Corporation, and other products from Sun Microsystems, Hewlett-Packard, and Silicon Graphics Inc. These packages offer levels of computer sharing and collaboration from videoconferencing and file sharing to full application sharing. Each implementation runs on specific operating systems. Although they use various underlying communication protocols, most recent designs work over local area and wide area networks (LANs/WANs), including the Internet. For example, the NetMeeting product provides conferencing tools like chat, whiteboard, file transfer, audio and videoconferencing, and non-real-time, selected-window 2-D application sharing over T120 protocols layered on the Transmission Control Protocol/Internet Protocol (TCP/IP).[4] NetMeeting runs only on Microsoft platforms (Windows 95 and Windows NT operating systems). The current products are deficient, however, in that they do not support multiple operating systems, do not operate in real time, and do not share 3-D graphics.

User Interface

In this section, we describe our choice of a simple user interface for the sharing area of a desktop and our design of the Shared Desktop Manager for client-server computing.

Many collaboration tools for sharing computer information (graphical desktop, keyboard, mouse, and audio of a given computer) were complete systems and required too much effort on the part of the users just to learn how to share information. A focus on learning collaboration tools often requires users to become experts in the collaboration software rather than in the applications that they may share. Since the various 3-D graphics packages that needed to be shared were complicated in themselves, we decided to implement a simple user interface in the Shared Desktop application that nearly all audiences could easily learn and use.

In the Shared Desktop design, we designated part of the desktop screen as a sharing area. Graphics objects such as icons and applications located within the sharing area can be accessed by all conference participants. To share a new application, a participant moves the application into the sharing area. To remove an application, a participant moves it outside the sharing area. If the sharing area encompasses the entire desktop of the initiating participant, all applications are shared. We used standard pull-down menus and widgets provided by either the UNIX X Motif toolkit or the Microsoft Windows libraries. We named the sharing area the "viewport"; it is viewed on the desktop as a user-defined area of rectangular size and location. Any graphical object placed into the viewport is marked as shareable with client users in a collaboration. We designed the viewport so that it is always on the bottom of a given stack of windows on a desktop. Thus, when Shared Desktop is minimized, so is its viewport. The objects that had been within the viewport are returned to the initiator's desktop and are no longer shared. With a quick minimization, the server collaborator can pause any sharing that was in progress without disconnecting from the client users.

Figure 1 illustrates a UNIX server with a Shared Desktop viewport connected to several client systems. The server's viewport contains no shared objects within its confines, and each client screen shows a viewport received from the server.

The viewport can be set to represent the entire visible desktop, or it can be set to equal only the size of a given application on the screen. Accordingly, a user who is acting as the server can determine how much of a given desktop to share among the client collaborators. The concept of a viewport is valuable because the principal collaborator (at the server) can quickly glance at the screen and determine what to capture and send to other participants. (The objects and applications sent from the server are designated by solid lines in Figure 1.) The Shared Desktop application requires no further action to set up an application for sharing.

Each client sends keyboard and mouse events to the server to control any application present in the viewport (remote control is shown as dashed lines in Figure 1). Server and clients synchronize cursor movements so that any conference member can watch as others make changes to a shared application. This allows the cursor to become a pointer during a session. Shared Desktop implements an "anarchy" form of remote control, with all mice and keyboards active simultaneously.

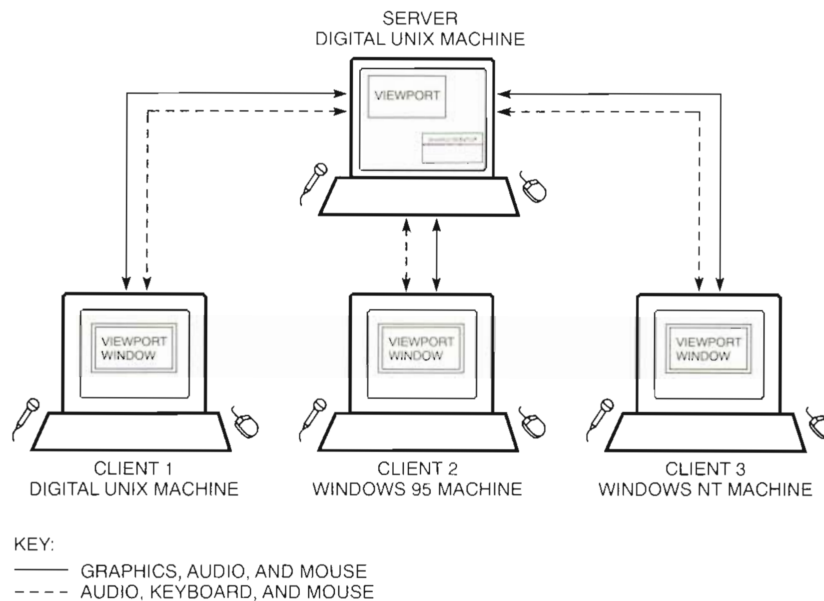


Figure 1
Server Desktop with Viewport and Clients

When a user initiates a collaboration, the audio is off by default but remains integral to a session as a convenience (as opposed to using the telephone). Through a pull-down menu operation, the server enables audio for all participants in one operation. The usual audio management tools used to set microphone recording levels and speaker/headset play-back levels are available. As Figure 1 indicates, the UNIX machine collects audio and distributes it to the three client collaborators. Likewise, the three clients collect audio and send it back to the server for mixing. In this way, all participants can hear one another and interact with whatever objects appear in the viewport on the server's screen.

Figure 2 shows the Shared Desktop Manager from the initiator's viewport running on the UNIX server. A participant may use a Session pull-down menu to control the viewport and to connect and disconnect other conference members. The Options menu allows for audio, remote cursor, and call-back control. The application's Help pull-down menu provides the usual help information similar to a Windows help facility or a Web browser's help. The window lists the status of attached clients.

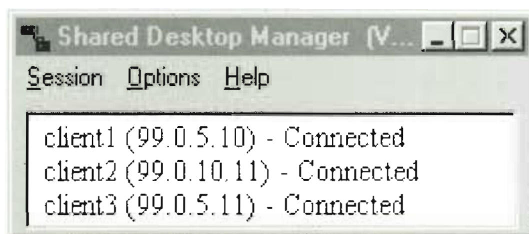


Figure 2
Shared Desktop Manager

Upon connection, participants can hear and interact with the server. The resultant audio dialogue combined with the graphics, keyboard, and mouse interactions facilitate a collaboration environment in which participants share an application. Since each user can operate a separate mouse and keyboard, the audio channel acts as a synchronization mechanism to indicate which collaborator controls the shared applications at any given moment. The participants communicate their actions verbally, interacting in much the same way as people who are sitting around a table and working.

Design Features

For our implementation, we concentrated on three principal areas: protocol splitting, screen capture and data handling, and dissimilar frame buffers. In this section, we discuss our investigation into using a protocol splitter and our decision to rely on screen capture and data handling. We also discuss dissimilar frame buffers.

Protocol Splitting

We looked for a way to distribute 3-D graphics among workstations and PCs that would be independent of the application, graphics protocol, architecture, operating system, and windowing system. On UNIX, we found application sharing provided by distributed windows protocols. For example, the X Protocol[5] allows a user to send an application to a nonlocal display and to send X applications protocol messages to several screens simultaneously. A protocol splitter, however, has disadvantages due to its requirements for bandwidth, programming, and latency.

Protocol splitters require distribution of graphics commands and display lists by means of a network. Three-dimensional models often contain megabytes of graphical information that describe specific screen operations. When displaying a model locally, these graphics operations move quickly and easily over system buses that are capable of handling hundreds of megabytes per second. However, when these same graphics objects are copied over computer networks, the amount of information can overload even the highest-speed networks. For example, using a 100-megabyte (MB) Pro/ENGINEER truck assembly, a current generation 3-D workstation can load, display, and rotate the truck once in approximately 2 minutes. The same operation between two identical 3-D workstations takes 20 minutes when performed by a distributed protocol, and the rotation of the truck does not appear fluid to the user. If the same data or application is duplicated on every machine, only updates with synchronizing events are distributed, but this requires that all machines have the same graphics hardware.

The programming software needed for interoperation among dissimilar operating and windowing systems using protocol splitting is quite involved. The ability to support X11 desktops, Windows 95 desktops, and Windows NT desktops while using multiple 3-D protocols like OpenGL and Direct3D would require that these protocols exist on all platforms.

Latency requirements for 3-D are very stringent. Thus, any network jitter makes even the best network link create breakup (visual distortions) when rotating 3-D objects. Network jitter also causes delays in sending window protocol messages; as a delay increases, the window events may no longer be useful. For example, when rotating a 3-D object, the delayed events must propagate as the network permits although this may once again congest the network since the events may no longer be needed. The object has now rotated to a new view. The ability to drop some protocol messages in a time-critical way is a requirement for collaborating with 3-D objects, and the protocol splitter approach to sharing has no solution for this problem.

Screen Capture and Data Handling

To overcome these issues, we investigated capturing the screen display, the final bitmap result of the interaction of graphics hardware and software that the viewer sees. Capturing the screen is in itself nothing new; it has been used for some time to include screen visuals in document preparation. Initially, we were skeptical that capturing the screen display could be a useful mechanism since the amount of data on a screen can be prodigious. Screen graphics depth and resolution can make the amount of data in any given graphics object very large. For example, for a 24-plane frame buffer with a 1,280 by 1,024 resolution, the total amount of data to capture would be (24 x 1,280 x 1,024)/8 or about 4 MB. Using the computational power of the Alpha microprocessor for reducing the data, we continued our investigation. We found that this approach requires the windowing system to perform screen capture by means of a non-CPU-intensive routine (direct memory access [DMA] as opposed to programmed I/O). Based on our tests, we concluded that screen capture technology would be easier to implement than a protocol splitter, would have better latency for 3-D operations than a protocol splitter, and would be easily adaptable to the various windowing systems and 3-D protocols we wished to have interoperate.

Graphics Compression The screen capture approach requires a number of steps to efficiently prepare the data for transmission. First, the contents of a viewport are captured, and the sample is saved for comparison with successive samples. Second, the captured viewport samples are differenced to find screen pixels that have not been changed and delta values for those that have been changed. Third, the resultant array of values is compressed by a fast, run-length encoding (RLE) of the array of difference samples. A more CPU-intensive compression may now be applied. The fourth step is to apply LZ77 compression that reduces the remaining RLE data to its smallest form. In step four, the original data has been reduced while retaining its characteristics so that it can be restored (uncompressed) without loss on a receiving computer. This final lossless stage of compression occurs only if it reduces the amount of data and if the network was busy during a previous transmission. Lossless compression is important for the nondestructive transfer of data from the server's screen to the clients' screens and has application in industry. As an example, consider a doctor who is sharing an x-ray with an out-of-town colleague. If the graphics were compromised by a lossy compressor, the collaborators could not be guaranteed that the transmitted x-ray was identical to the one sent. With the Shared Desktop application, the doctor who is sending the x-ray is guaranteed that the original graphics are restored on the colleague's display. In some forms of compression, data is thrown out by the algorithm and never restored, so that the final screen data may not accurately reflect the original graphics. Figure 3 shows the steps in the capture and compression sequence.

On the Alpha architecture, these compression steps are performed as 64-bit operations, both in the data manipulation and the compression algorithms. The Alpha architecture lends itself to a fast and efficient implementation of the algorithms, so that the capture of the viewport and the multistage compression of the data can be accomplished in real time. Approximately half of the number of instructions is used on a processor that is twice as fast as a 32-bit architecture. In addition to its 64-bit routine, the RLE is implemented as a 32-bit routine and as a comparison routine.

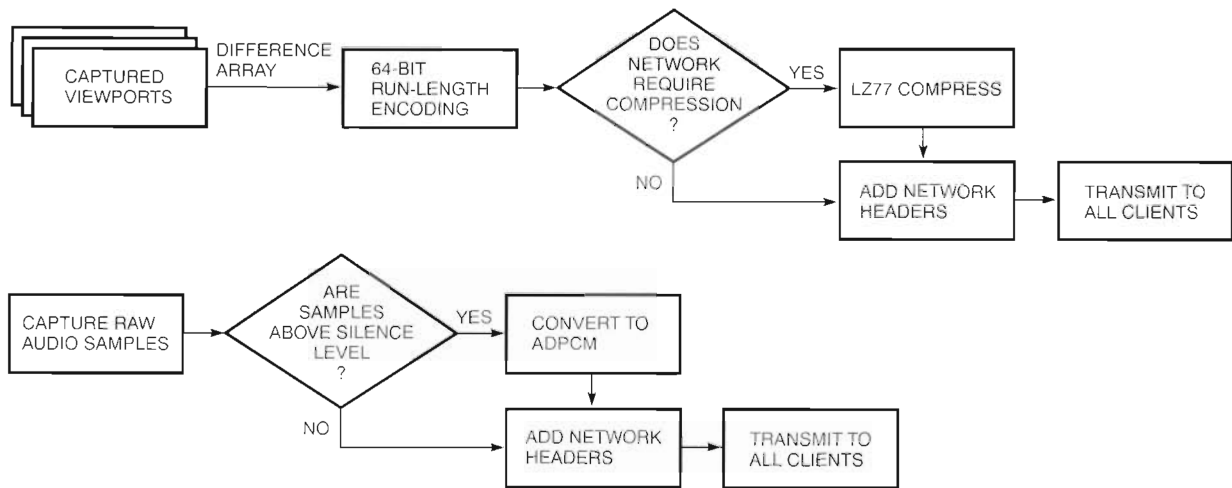


Figure 3
Graphics and Audio Compression Data Flow Diagram

Audio Compression Similar to the graphics compression described, the audio compression in Shared Desktop involves several steps. First, the audio samples are captured through a microphone and sound card combination. These samples are compared with the background noise level (determined prior to beginning a conference) to see if the samples are useful. Samples below the background noise level are not transferred. This implements a silence detection method whereby only useful samples will advance to the next level of compression. Second, the next compression uses G.711 or other similar audio compression standards and converts adaptive differential pulse code modulation (ADPCM) samples at 64 kilobits per second into 16 kilobits per second (4:1 lossy compression).[6] Third, this data is then ready for transfer to a receiving computer so that it may be decompressed and output to a speaker or a headset. The audio stream resulting from these steps generates at most 16 kilobits per second when someone is speaking, and no output when it is silent. Figure 3 also shows the audio compression steps.

Data Transmission After the graphics and audio data are collected and compressed, they are combined and transmitted across the network by a patented, higher-level protocol that ensures timely delivery of each packet.[7] All packets are sent using TCP/IP over the Internet. Although the higher-level protocol does not guarantee true real-time characteristics, the patented protocol allows for coherent audio, synchronization of graphics and cursor events, and near real-time graphics animation.

As an example, the screen capture shown in Figure 4 displays a 100-MB Pro/ENGINEER assembly being shared through the Shared Desktop application. The Shared Desktop Manager system (system where the assembly database resides) is an AlphaStation 500 workstation running the DIGITAL UNIX operating system with a PowerStorm 4D60 graphics controller. In this example, an 800- by 600-pixel by 24-bit Shared Desktop viewport is being captured, compressed, and transmitted to the Shared Desktop client system at about five updates per second. The update rate is determined by the capture viewport size, the extent of detail changes between captures, the amount of processing power needed by the application to make changes to the model, and the speed of the network. In this example, when rotating the truck assembly, a compressed stream of 400 to 500 kilobytes per second is generated and represents the five updates per second mentioned. A simple assembly might be able to do a rotation with Shared Desktop capturing and transmitting 15 updates per second, and a more complicated model (like the truck assembly shown) would receive fewer updates per second.

Dissimilar Frame Buffers

To complete the requirements of our implementation, we needed to share graphics information across dissimilar hardware, i.e., machines with different graphic frame buffer depths. The frame buffer depth refers to the amount of storage the graphics adapter gives to each displayed pixel on the screen. A 16-bit-deep display assigns each pixel a 16-bit value to represent the pixel. This representation is usually the color information for the pixel, i.e., what color the user sees for a given pixel.[8] The frame buffer depths are a necessary reality since different graphics devices have widely varying screen depths, ranging from 4 planes (4 bits per pixel) to 32 planes (32 bits per pixel). Typically, higher end graphics devices have higher-depth graphics outputs, especially for 3-D graphics, and the lower-

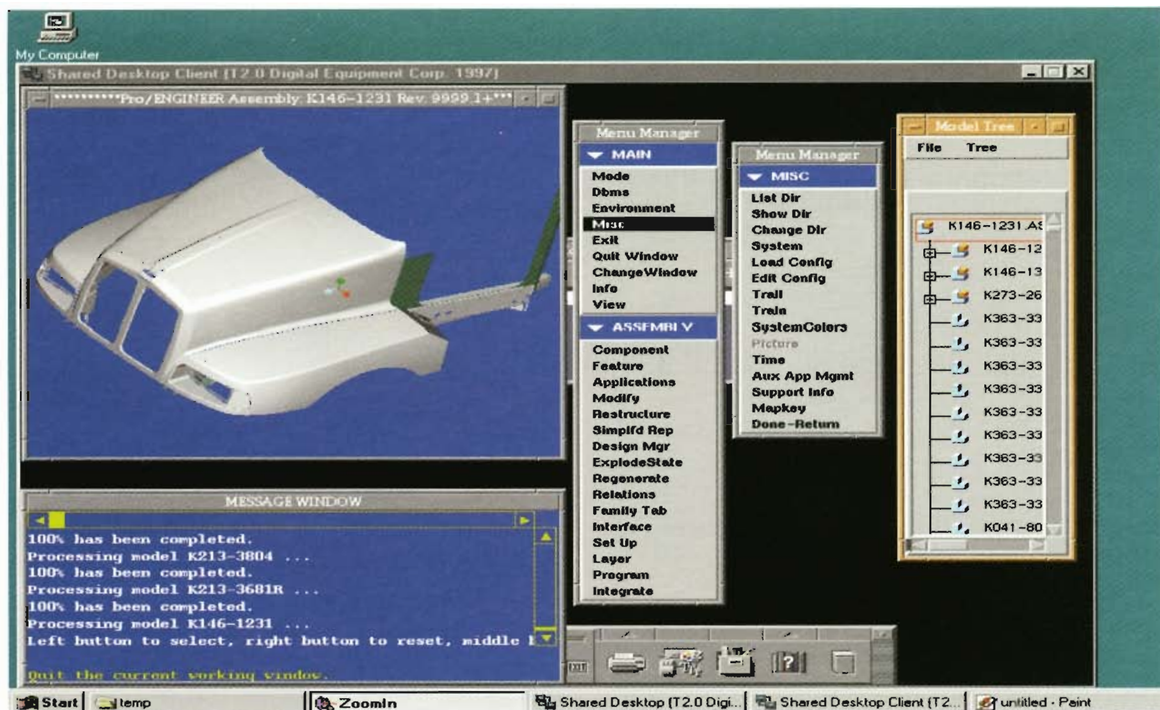


Figure 4
Screen Capture of the Windows Shared Desktop Client Sharing Pro/ENGINEER with a UNIX Shared Desktop Server

depth displays are usually found on less-capable, 2-D graphics platforms. Most laptop computers have low bit depth (8 to 16) displays and no 3-D capabilities. Commodity PCs also typically have 8- or 16-plane depths. Graphics devices that support 3-D graphics provide deeper display types such as 24-bit or 32-bit. Some devices support a mix of several or all the bit depths listed in the matrix (below) either concurrently or for the entire screen at one time.

We defined a matrix of screen depths and proceeded to fill in the various combinations so that the application would work effectively across different platforms and graphics hardware capabilities. The matrix enables computers without 3-D capability to display the output from 3-D-capable graphics devices. The matrix of screen-depth combinations follows.

Output Bitmap Depth	Input Screen or Visual Type Depth						
	4	8	12	15	16	24	32
4	x	md	md	d	d	d	d
8	e	mx	md	d	d	d	d
12	n	n	n	n	n	n	n
15	e	me	me	x	d	d	d
16	e	me	me	e	x	d	d
24	e	me	me	e	e	x	d
32	e	me	me	e	e	e	x

The matrix shows input screen or visual type depth across the top row and delineates output bitmap depth on the left column. Bitmap depths of 4, 8, 15, 16, 24, and 32 are used in Windows systems, and depths of 4, 8, 12, 24, and 32 are used in X11. The *x* in the matrix requires no conversion and is captured and displayed without the need for additional conversion. The *e* shows bitmap depths that can be expanded to the output format by using a colormap or by shifting pixels into the correct format. The *d* shows that information must be dithered to match the output. Dithering can result in a minimal loss of information, but we have developed a very good and efficient method of doing this conversion. The *m* (mix mode) marks those visual types on X11 that can exist on the screen when the root depth is 24 or 32; i.e., an 8-bit window can be present on a 24-bit display. The mix mode requires a different interpretation of the 24-bit pixels prior to compression and transmission. Since no 12-bit output displays exist, *n* marks inapplicable transformations. Alternate formats of 24 pixels (3 bytes per pixel and blue/green/red [BGR] triples) are supported as well as 8-bit pseudocolor and 8-bit true color.

Sample Uses

Like other collaboration software, the Shared Desktop application can be used in remote situations to help

people communicate and share data. These uses include telecommuting, debugging/support, and education.

Telecommuting

One feature we built into Shared Desktop is the ability to originate a sharing session from a remote location. Our intent was to allow an individual to work outside the office environment on a home PC or a laptop computer. In the telecommuting scenario, a workstation with high-end graphics functions and applications located in the office would call back the home user's low-end system and present the user with his work environment. For example, consider a user of PTC's Pro/ENGINEER who is working on a 3-D assembly with a 100-MB database and must make a change to the part from home. Prior to the Shared Desktop application, the only options were either to mimic the work environment at home or drive to the office to make the change. To mimic a work environment, the equipment at home must support Pro/ENGINEER software and might require 3-D hardware. In addition, the user would have to retrieve a recent version of the 100-MB database over the telephone lines, which would take many hours to copy. With the Shared Desktop application, the user can access the 100-MB part using the low-end computer over standard telephone lines. The changes to the assembly then occur on the system and to the large database at the office.

Remote Debugging/Support

Another use of the Shared Desktop application is for customer support or remote debugging. Consider the user of a 3-D design application who discovers a bug in a new version of the software. A complex model often causes a bug that requires software support to obtain the database to re-create the problem. Using Shared Desktop, a user could show a support representative the problem on the running application, as opposed to filing a problem report.

Off-site Training

A remote training scenario provides a final example of collaboration using computers. The Shared Desktop application facilitates remote training by connecting students in a sharing session. Each student's desktop displays a lesson composed of the course material installed on the instructor's desktop. Students interact with the teacher by audio, mouse, and keyboard actions on objects in the screen viewport. In essence, the teacher uses the synchronized cursors to highlight or point to objects on the screen.

Conclusion and Future Directions

The Shared Desktop collaboration software employs a simple user interface that emphasizes ease of 3-D application sharing and audio conferencing. Compared to application sharing based on a protocol splitter, the Shared Desktop application offers easier interoperability and better latency during 3-D operations. With a protocol splitter approach, it is difficult to decide which, if any, graphics events to drop when network jitter or network bandwidth delays occur. Our approach is synchronized to the last screen capture. When the network is no longer congested, the current screen capture can be sent, thus minimizing the perceived effect of the network delay. The only disadvantage to bitmap sharing is its requirement that the windowing system and display driver implement a DMA screen capture function and not programmed I/O. DMA screen capture requests have a minimal load on the windowing system.

We are planning a number of improvements to the advanced development version of Shared Desktop. In our initial work, we made no changes to the windowing systems. Ideally, the product version might have a mechanism that notifies an application when and where another application has made changes to the screen. With the added ability to capture only those areas of the screen that have changed since the last notification, the windowing system could perform the first two steps in the capture process.

Although the compression scheme we implemented works for most cases, some graphics may not compress well using the combination of RLE and LZ77. Instead, content-specific compression or adaptive compression techniques might be better applied. This is an area of study we hope to pursue.

The current graphical user interface (GUI) lacks some conferencing features. The product version will be packaged with other applications to provide video, chat, whiteboard, file transfer, and user locator/directory services.

Finally, the sharing model we implemented for the Shared Desktop application is easily ported to other systems. Thus the application could be available for widespread use.

References

1. A. Hopper, "Pandora—An Experimental System for Multimedia Applications," *Operating Systems Review*, vol. 24, no. 2 (1990): 19–35.

2. J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide* (Reading, Mass.: Addison-Wesley Publishing Company, 1993).

3. *Visual C++ Version 4.0 SDK Programmer's Guide RISC Edition* (Redmond, Wash.: Microsoft Corporation, 1995).

4. *Multipoint Still Image and Annotation Protocol*, ITU-T Recommendation T.126 (Draft) (Geneva: International Telegraph and Telephone Consultative Committee, 1997).

5. R. Scheifler, J. Gettys, R. Newman, A. Mento, and A. Wojtas, *X Window System: C Library and Protocol Reference* (Burlington, Mass.: Digital Press, 1990).

6. *Pulse Code Modulation (PCM) of Voice Frequencies*. CCITT Recommendation G.711 (Geneva: International Telecommunications Union, 1972).

7. R. Palmer and L. Palmer, "Video Teleconferencing for Networked Workstations," United States Patent no. 5,375,068 (1994).

8. P. Heckbert, "Color Image Quantization for Frame Buffer Display," *Computer Graphics*, vol. 16, no. 3 (1982).

Biographies

Lawrence G. Palmer
Larry Palmer is a consulting engineer in Workstations Graphics Development. He joined DIGITAL in 1984 and currently co-leads the Shared Desktop project. He holds a B.S. in chemistry (summa cum laude) from the University of Oklahoma and belongs to Phi Beta Kappa. He is co-inventor for nine patents on enabling software technology for audio-video teleconferencing.

Ricky S. Palmer
Ricky Palmer is a consulting engineer in Workstations Graphics Development. He joined DIGITAL in 1984 and currently co-leads the Shared Desktop project. He holds a B.S. in physics (magna cum laude), a B.S. in mathematics, and an M.S. in physics from the University of Oklahoma. He is co-inventor for nine patents on enabling software technology for audio-video teleconferencing.

David C. P. LaFrance-Linden

Challenges in Designing an HPF Debugger

High Performance Fortran (HPF) provides directive-based data-parallel extensions to Fortran 90. To achieve parallelism, DIGITAL's HPF compiler transforms a user's program to run as several intercommunicating processes. The ultimate goal of an HPF debugger is to present the user with a single source-level view of the program at the control flow and data levels. Since pieces of the program are running in several different processes, the task is to reconstruct the single control and data views. This paper presents several of the challenges involved and how an experimental debugging technology, code-named Aardvark, successfully addresses many of them.

As we learn better ways to express our thoughts in the form of computer programs and to take better advantage of hardware resources, we incorporate these ideas and paradigms into the programming languages we use. Fortran 90[1,2] provides mechanisms to operate directly on arrays, e.g., `A=2*A` to double each element of `A` independent of rank, rather than requiring the programmer to operate on individual elements within nested `DO` loops. Many of these mechanisms are naturally data parallel. High Performance Fortran (HPF)[3,4] extends Fortran 90 with data distribution directives to facilitate computations done in parallel. Debuggers, in turn, need to be enhanced to keep pace with new features of the languages. The fundamental user requirement, however, remains the same: Present the control flow of the program and its data in terms of the original source, independent of what the compiler has done or what is happening in the run-time support. Since HPF compilers automatically distribute data and computation, thereby widening the gap between actual execution and original source, meeting this requirement is both more important and more difficult.

This paper describes several of the challenges HPF creates for a debugger and how an experimental debugging technology, internally code-named Aardvark, successfully addresses many of them using techniques that have applicability beyond HPF. For example, programming paradigms common to explicit message-passing systems such as the Message Passing Interface (MPI)[5-7] can benefit from Aardvark's methods.

The HPF compiler and run time used is DIGITAL's HPF compiler,[8] which produces an executable that uses the run-time support of DIGITAL's Parallel Software Environment.[9] DIGITAL's HPF compiler transforms a program to run as several intercommunicating processes. The fundamental requirement, then, is to give the appearance of a single control flow and a single data space, even though there are several individual control flows and the data has been distributed. In the paper, I introduce the concept of logical entities and show how they address many of the control flow challenges. A discussion of a rich and flexible data model that easily handles distributed data follows. I then point out difficulties imposed on user interfaces, especially when the program is not in a completely

consistent state, and indicate how they can be overcome. Sections on related work and the applicability of logical entities to other areas conclude the paper.

Logical Entities

From the programmer's perspective, an HPF program is a single process/thread with a single control flow represented by a single call stack consisting of single stack frames. A debugger should strive to present the program in terms of these single entities. A key enabling concept in the Aardvark debugger is the definition of logical entities in addition to traditional physical entities. Generally, a *logical entity* collects several physical entities into a single entity. Many parts of Aardvark are unaware of whether or not an entity is logical or physical, and a debugger's user interface uses logical entities to present program state.

A *physical entity* is something that exists somewhere outside the debugger. A physical process exists within the operating system and has memory that can be read and written. A physical thread has registers and (through registers and process memory) a call stack. A physical stack frame has a program counter, a caller stack frame, and a callee stack frame. Each of these has a representation within the debugger, but the actual entity exists outside the debugger.

A logical entity is an abstraction that exists within the debugger. Logical entities generally group together several related physical entities and synthesize a single behavior from them. In C++ terms, a process is an abstract base class; physical and logical processes are derived classes. A logical process contains as data members a set of other (probably physical) processes. The methods of a logical process, e.g., to set a breakpoint, bring about the desired operations using logical algorithms rather than physical algorithms. The logical algorithms often work by invoking the same operation on the physical entities and constructing a logical entity from the physical pieces. This implies that some operations on physical entities can be done in isolation from their logical containers. Aardvark makes a stronger statement: Physical entities are the building blocks for logical entities and are first-class objects in their own right. This allows physical entities to be used for traditional debugging without any additional structure.[10]

A positive consequence of this object-oriented design is that a user interface can often be unaware of the physical or logical nature of the entities it is managing. For example, it can set a breakpoint in a process or navigate a thread's stack by calling virtual methods declared on the base classes.

Some interesting design questions arise: What is a process? What is a thread? What is a stack frame? What operations are expected to work on all kinds of processes but actually only work on physical processes? Experience to date is inconclusive. Aardvark currently defines the base classes and methods for logical entities to include many things that are probably specific to physical entities. This design was done largely for convenience.

Sometimes a logical entity is little more than a container of physical entities. A logical stack frame for threads that are in unrelated functions simply collects the unrelated physical stack frames. Nevertheless, logical stack frames provide a consistent mechanism for collecting physical stack frames, and variants of logical stack frames can discriminate how coordinated the physical threads are. The concept of logical entities does not apply to all cases, though. Variables have values, and there does not seem to be anything logical or physical about values. Yet, if a replicated variable's values on different processors are different, there is no single value and some mechanism is needed. Rather than define logical values, Aardvark provides a differing values mechanism, which is discussed in a later section of the same name.

Controlling an HPF Process

Users want to be able to start and stop HPF programs, set breakpoints, and single step. From a user interface and the higher levels of Aardvark, these tasks are simple to accomplish—ask the process or thread, which happens to be logical, to perform the operation. Within the logical process or thread, however, the complexity varies, depending on the operation.

Starting and Stopping

Starting and stopping a logical thread is straightforward: Start or stop each component physical thread. Some race conditions require care in coding, though. For example, starting a logical thread theoretically starts all the corresponding physical threads simultaneously. In practice, Aardvark serializes the physical threads. In Figure 1, when all the physical threads stop, the logical thread is declared to be stopped. Aardvark then starts the logical thread at time "+" and proceeds to start each physical thread. Suppose the first physical thread (thread 0) stops immediately, at time "*." It might appear that the logical thread is now stopped because each physical thread is stopped. This scenario does not take into account that the other physical threads have not yet been started. Timestamping execution state transitions, i.e., ordering the events as observed by Aardvark, works well; a logical thread becomes stopped only when all its physical threads have stopped after the time that the logical thread was started. An added complexity is that some reasons for stopping a physical thread should stop the other physical threads and the logical thread. In this case, pending starts should be cancelled.

Breakpoints

Setting a breakpoint in a logical process sets a breakpoint in each physical process and collects the physical

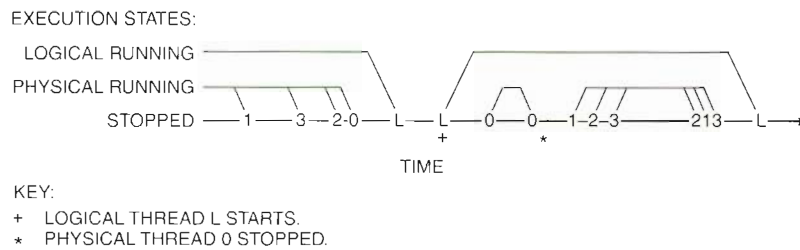


Figure 1
Determining When a Logical Thread Stops

breakpoint representations into a logical breakpoint. For HPF, any action or conditional expression is associated with the logical breakpoint, not with the physical breakpoints. Consider the expression `ARRAY(3,4).LT.5`. Even if the element is stored in only one process, the entire HPF process needs to stop before the expression is evaluated; otherwise, there is the potential for incorrect data to be read or for processes to continue running when they should not. This requires each physical process to reach its physical breakpoint before the expression can be evaluated. Once evaluated, the process remains stopped or continues, depending on the result. For HPF, a breakpoint in a logical process implies a global barrier of the physical processes.

Recognizing and processing a thread reaching a logical breakpoint is somewhat involved. Aardvark's general mechanism for breakpoint determination is to ask the thread's operating system model if the initial stop reason could be a breakpoint. If this is the case, the operating system model provides a comparison key for further processing.

For physical DIGITAL UNIX threads, a `SIGTRAP` signal could be a breakpoint, with the comparison key being the program counter address of the potential breakpoint instruction. This comparison key is then used to search the breakpoints installed in the physical process to determine which (if any) breakpoint was reached. If a breakpoint was reached, the stop reason is updated to be "stopped at breakpoint." All this physical processing happens before the logical algorithms have a chance to notice that the physical thread has stopped. Therefore, by the time Aardvark determines that a logical thread has stopped, any physical threads that are stopped at a breakpoint have had their stop reasons updated.

For a logical thread, the initial (logical) stop reason could be a breakpoint if each of the physical threads is stopped at a breakpoint, as shown in Figure 2. The comparison key in this case is the logical stop reason itself. The breakpoints of the component stop reasons are then compared to the component breakpoints of the installed logical breakpoints to determine if a logical breakpoint was reached. If there is a match, the logical thread's stop reason is updated.

Aardvark achieves the flexibility of vastly different types of comparison keys (machine addresses and logical stop reasons) by having the comparison key type be the most basic Aardvark base class, which is the equivalent of Java's `Object` class, and by using run-time typing as necessary.

Single Stepping

Single stepping a logical thread is accomplished by single stepping the physical threads. It is not sufficient to single step the first thread, wait for it to stop, and then proceed with the other threads. If the program statement requires communication, then the entire HPF program needs to be running to bring about the communication. This implies that single stepping is a two-part process—initiate and wait—and that the initiation mechanism must be part of the exposed interface of threads.

As background, running a thread in Aardvark involves continuing the thread with a *run reason*. The

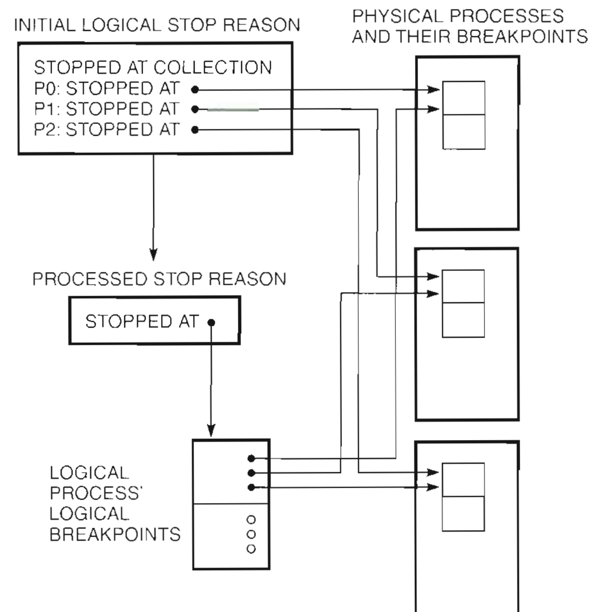


Figure 2
Logical Breakpoint Determination

run reason is empowered to take the actions (e.g., setting or enabling temporary breakpoints) necessary to carry out its task. In this paper, the word *empowered* means that the reason has a method that will be called to do reason-specific actions to accomplish the reason's semantics. This relieves user interfaces and other clients from figuring out how to accomplish tasks. As a result, Aardvark defines a "get single-stepping run reason" method for threads. Clients use the resulting run reason to continue the thread, thereby initiating the single-step operation.

Therefore, single stepping a logical thread in Aardvark involves calling the (logical) thread's "get single-stepping run reason" method, continuing the thread with the result, and waiting for the thread to stop. The "get single-stepping run reason" method for a logical thread in turn calls the "get single-stepping run reason" method of the component (physical) threads and collects the (physical) results into a logical single-stepping run reason. When invoked, the logical reason continues each physical thread with its corresponding physical reason.

Single stepping dramatically demonstrates the autonomy of the physical entities. When continuing a (logical) thread with a (logical) single-stepping run reason, the physical threads can start, stop, and be continued asynchronously to each other and without any intervention from a user interface, the logical entities, or other clients. This is especially true if the thread was stopped at a breakpoint. In this case, continuing a physical thread involves replacing the original instruction, machine single stepping, putting back the breakpoint instruction, and then continuing with the original run reason. Empowering run reasons (and stop reasons) to effect the necessary state transitions enables physical entities to be autonomous, thus relieving the logical algorithms from enormous potential complexity.

Coordinating Physical Entities

The previous discussion describes some logical algorithms. The section "Starting and Stopping" describes using timestamps to determine when a logical thread becomes stopped (see Figure 1), and the section "Breakpoints" describes a logical thread possibly reaching a breakpoint (see Figure 2). The physical entities need to be coordinated so that the logical algorithms can be run. In Aardvark, this is done with a *process change handler*. A process change handler is a set of callbacks that a client registers with a process and its threads, allowing the client to be notified of state changes. For example, if a user interface is notified that a thread has stopped and that the reason is a UNIX signal, the user interface can look up the signal in a table to determine if it should continue the thread (possibly discarding the actual signal) or if it should keep the thread stopped.

In the context of HPF, a user interface registers its process change handler with the logical HPF process. During construction of the logical process, Aardvark registers a *physical-to-logical process change handler* with the physical processes. It is this physical-to-logical handler that coordinates the physical entities. When the first physical thread stops, as at time "*" in Figure 1, the handler is notified but notices that the timestamps do not indicate that the logical thread should be considered to have stopped. When the last physical thread stops, the handler then synthesizes a "stopped at collection" logical stop reason, as in Figure 2, and informs the (logical) thread that it has stopped.

Aardvark defines some callbacks in process change handlers that are for HPF and other logical paradigms. These callbacks allow a user interface to implement policies when a thread or process goes into an intermediate state. For example, at time "*" in Figure 1 a physical thread has stopped but the logical thread is not yet stopped. Whenever a physical thread stops, the handler's "component thread stopped" callback is invoked. A possible user interface policy is[11]

- If the component thread stopped for a *nasty* reason, such as an arithmetic error, try to stop all the other component threads immediately in order to minimize divergence among the physical entities.
- If this is the first component thread that stopped for a *nice* reason, such as reaching a breakpoint, start a timer to wait for the other component threads to stop. If the timer goes off before all the other component threads have stopped, try to stop them because it looks like they are not going to stop on their own.
- If this is the last component thread, cancel any timers.

The user interface can provide the means for the user to define the timer interval, as well as other attributes of policies. These policies and their control mechanisms are not the responsibility of the debug engine.

Examining an HPF Call Stack

When an HPF program stops, the user wants to see a call stack that appears to be a single thread of control. Sometimes this is not possible, but even in those cases, a debugger can offer a fair amount of assistance. The HPF language provides some mechanisms that also need to be considered. The `EXTRINSIC(HPF_LOCAL)` procedure type allows procedures written in Fortran 90 to operate on the local portion of distributed data. This type is useful for computational kernels that cannot be expressed in a data-parallel fashion and do not require communication. The `EXTRINSIC(HPF_SERIAL)` procedure type allows data to be mapped to a single process that runs the procedure. This type is useful for calling inherently serial code, including user interfaces,

which may not be written in Fortran. DIGITAL's HPF compiler also supports *twinning*, which allows serial code to call parallel HPF code. All these mechanisms affect the call stack or how a user navigates the call stack. They require underlying support from the debugger as well as user interface support.

Logical Stack Frames

Aardvark's logical entity model applies to stack frames: logical stack frames collect several physical stack frames and present a synthesized view of the (logical) call stack. Currently, Aardvark defines four types of logical stack frames to represent different scenarios that can be encountered:

1. Scalar, in which only one physical thread is semantically active
2. Synchronized, in which all the threads are at the same place in the same function
3. Unsynchronized, in which all the threads are in the same function but at different places
4. Multi, in which no discernible relationship exists between the corresponding physical threads

Aardvark's task is to discover the proper alignment of the physical frames of the physical threads, determine which variant of logical frame to use in each case, and link them together into a call stack. Ideally, all logical frames are synchronized, which means that the program is in a well-defined state. This is true most of the time with HPF; the Single Program Multiple Data (SPMD) nature of HPF causes all threads to make the same procedure calls from the same place, and breakpoints are barriers causing the threads to stop at the same place.

Aardvark's alignment process starts at the outermost stack frames of the physical threads (the ones near the Fortran `PROGRAM` unit) and then progressively examines the callees (toward where the program stopped). Starting from the innermost frames is an error-prone approach. If the innermost frames are in different functions, Aardvark might construct a multiframe when the frames are actually misaligned because the physical stacks have different depths. As discussed in the section on twinning, depth is not a reliable alignment mechanism either. Starting at the outermost frames follows the temporal order of calls and also correctly handles recursive procedures. The disadvantage of starting at the outermost frames is that each physical thread's entire stack must be determined before the logical stack can be constructed. Usually the programmer only wants the innermost few frames, so time delays in the construction process can reduce the ease of use of the debugger.[12]

Much of the time, the physical stack frames are at the same place because the SPMD nature of HPF causes the physical threads to have the same control flow. When a procedure is called, each thread executes the call and executes it from the same place. A logical breakpoint is reached when the physical threads are stopped at the same place at the corresponding physical breakpoints. These cases lead to synchronized frames. The most common cause of an unsynchronized frame is interrupting the program during a computation. Even in this case, the divergence is usually not very large. One reason for a multiframe is the interruption of the program while it is communicating data between processes. In this case, the code paths can easily diverge, depending on which threads are sending, which are receiving, and how much data needs to be moved. Scalar frames are created because of the semantic flow of the program: the main program unit is written in either a serial language or an HPF procedure called an `EXTRINSIC(HPF_SERIAL)` procedure type.

The result of the alignment algorithm is a set of frames collected into a call stack. The normal navigation operations (e.g., up and down) apply. Variable lookup and expression evaluation work as expected, also. Variable lookup works best for synchronized frames and, for HPF, works for unsynchronized frames as well. For multiframes, variable lookup generally fails because a variable name `VAR` may resolve to different program variables in the corresponding physical frames or may not resolve to anything at all in some frames. This failure is not because of a lack of information from the compiler but rather because multiframes are generally not a context in which a string `VAR` has a well-defined semantic.

Experience to date suggests that multiframes are of interest largely to the people developing the run-time support for data motion. Nevertheless, the point of transition from synchronized to unsynchronized to multi tells the user where control flows diverged, and this information can be very valuable.

Narrowing Focus

Using the previously mentioned techniques sometimes results in a cluttered view of the state of the entire program and difficulty in finding relevant information. Aardvark provides two ways to help. The first aid is a Boolean *focus* mask that selects a subset of the processes and then re-applies the logical algorithms. For properly chosen subsets, this can turn a stack trace with many multiframes into a stack trace with synchronized frames. A narrowed focus can also look behind the scenes of the twinning mechanism described in the next paragraph. The second aid is to view a single physical process in isolation, effectively turning off the parallel debugging algorithms. This technique is useful for debugging `EXTRINSIC(HPF_LOCAL)` and `EXTRINSIC(HPF_SERIAL)` procedures. The ability to retrieve the physical processes from a logical process is the major item that enables viewing a process in isolation; as mentioned before, physical entities are first-class objects.

Twinning

DIGITAL's HPF provides a feature called twinning in which a scalar procedure can call a parallel HPF procedure. This allows, for example, the main program consisting of a user interface and associated graphics to be written in C and have Fortran/HPF do the numerical computations. The feature is called twinning because each Fortran procedure is compiled twice. The *scalar twin* is called from scalar code on a designated process. Its duties include instructing the other processes to call the scalar twin, distributing its scalar arguments according to the HPF directives, calling the *HPF twin* from all processes, distributing the parallel data back onto the designated process after the HPF twin returns, and finally returning to its caller. The HPF twin is called on all processes with distributed data and executes the user-supplied body of the procedure.

At the run-time level, the program's entry point is normally called on a designated process (process 0), and the other processes enter a dispatch loop waiting for instructions. Conceptually, such a program starts in scalar mode and at some point transitions into parallel mode. An HPF debugger should represent this transition. Aardvark accomplishes this by having knowledge of the HPF twinning mechanism. When it notices physical threads entering the dispatch loop, Aardvark creates a scalar logical frame corresponding to the physical frame on process 0. It then processes procedure calls on process 0 only, creating more scalar frames, until it notices that the program transitions from scalar to parallel. This transition happens when all processes call the same (scalar twin) procedure: process 0 does so as a result of normal procedure calls; processes other than 0 do so from their dispatch loops. At this point, a logical frame is constructed that will likely be synchronized, and the frame processing described previously applies. The result is the one desired: a scalar program transitions to a parallel one.

DIGITAL's HPF goes a step further: it allows `EXTRINSIC(HPF_SERIAL)` procedures to call HPF code by means of the twinning mechanism. When an `EXTRINSIC(HPF_SERIAL)` procedure is called, processes other than 0 call the dispatch loop. When the scalar code on process 0 calls the scalar twin, the other processes are in the necessary dispatch loop. Aardvark tracks these calls in the same way as in the previous paragraph, noticing that processes other than 0 have called the dispatch loop and eventually call a scalar twin.

User Interface Implications

User interfaces and other clients must be keenly aware of the concept of logical frames and the different types of logical frames. Depending on the type of frame, some operations, such as obtaining the function name or the line number, may not be valid. Nevertheless, a user interface can provide useful information about the state of the program. The program used for the following discussion has a serial user interface written in C and uses twinning to call a parallel HPF procedure named `HPF_FILL_IN_DATA` (see Figure 3). The HPF procedure uses a function named `MANDEL_VAL` as a non-data-parallel computational kernel. The program was run on five processes. (Twinning is a DIGITAL extension. Most HPF programs are written entirely in HPF. This example, which uses twinning, was chosen to demonstrate the broader problem.)

Figure 4 shows the program interrupted during computation. Line 2 of the figure contains a single function name, `MANDEL_VAL`. Line 3 contains the function's source file name but lists five line numbers, implying that this is an unsynchronized frame. In fact, the user interface discovered that Aardvark created an unsynchronized logical frame. Instead of trying to get a single line number, the user interface retrieved the set of line numbers and presented them. In lines 4 through 10, the user interface also presented the range of source lines encompassing the lines of all the component processes. This user interface's `up` command (line 21) navigates to the calling frame. In this example, the frame is synchronized, causing the user interface to present the function's source file and single line number (line 26), followed by the single source file line (line 27).

Figure 5 shows a summary of the program's call stack when it was interrupted during computation. The summary is a mix of unsynchronized, synchronized, and scalar frames. Frame #0 (line 2) is unsynchronized, and the various line numbers are presented. Its caller, frame #1 (line 3), is synchronized with a single line number. All this is consistent with the previous discussion. Frame #1 is the HPF twin of the scalar twin in frame #2. The scalar twin of frame #2 is expected to be called by scalar code, confirmed by frames #3 and #4. Frame #5 is part of the twinning mechanism; process 0 is at line 499, while the other processors are all at line 506.

Narrowing the focus to exclude process 0 shows a different call stack summary (lines 9 through 16 of Figure 5). The new frame #0 (line 11) continues to be unsynchronized, but all the other frames are synchronized. The twinning dispatch loop (line 14) replaces the scalar frames of the global focus (lines 5 and 6). This replacement causes the new call stack, corresponding more closely to the physical threads, to have fewer frames than the global call stack.

Interrupting the program while idle within the user interface shows more about twinning and also shows a multiframe (see Figure 6). Most of the frames are scalar except for the twinning mechanism (frame #7, line 9) and the initial run-time frame (frame #8, line 10). Narrowing the focus to exclude process 0 shows the twinning mechanism while waiting. The twinning

```
subroutine hpf_fill_in_data(target, w, h, ccr, cci, cstep, nmin, nmax)
      integer, intent(in)      :: w, h
      byte, intent(out)        :: target(w,h)
      real*8, intent(in)       :: ccr, cci, cstep
      integer, intent(in)      :: nmin, nmax
!hpf$ distribute target(*, cyclic)

      integer :: cx, cy
      cx = w/2
      cy = h/2

      forall(ix = 1:w, iy = 1:h)                                        &
        target(ix,iy) = mandel_val(CMPLX(ccr + ((ix-cx)*cstep),        &
                                         cci + ((iy-cx)*cstep),        &
                                         KIND=KIND(0.0D0)),            &
                                   nmin, nmax)

contains

    pure byte function mandel_val(x, nmin, nmax)
      complex(KIND=KIND(0.0D0)), intent(in) :: x
      integer, intent(in)                   :: nmin, nmax

      integer                          :: n

      real(kind=KIND(0.0D0)) :: xorgr, xorgi, xr, xi, xr2, xi2, rad2
      logical                :: keepgoing

      n = -1
      xorgr = REAL(x)
      xorgi = AIMAG(x)
      xr = xorgr
      xi = xorgi

      do
         n = n + 1
         xr2 = xr*xr
         xi2 = xi*xi
         xi = 2*(xr*xi) + xorgi
         keepgoing = n < nmax
         rad2 = xr2 + xi2
         xr = xr2 - xi2 + xorgr
         if (keepgoing .AND. (rad2 <= 4.0)) cycle
         exit
      end do

      if (n >= nmax) then
         mandel_val = nmax-nmin
      else
         mandel_val = MOD(n, nmax-nmin)
      end if

    end function mandel_val

end subroutine hpf_fill_in_data
```

Figure 3
HPF_FILL_IN_DATA Procedure (Source Code for Figures 4 and 5)

mechanism at frames #5 and #6 (lines 23 and 24) is similar to the mechanism at frames #3 and #4 (lines 14 and 15) of Figure 5. In Figure 6, they do not call a scalar twin but rather call the messaging library to receive instructions from process 0. The messaging library, however, is often not synchronized among the peers, and frame #2 (line 15) shows a multiframe. This user interface shows a multiframe as a collection of one-line summaries of the physical frames (lines 16 through 20).

Examining HPF Data

Examining data generally involves determining where the data is stored, fetching the data, and then presenting it. HPF presents difficulties in all three areas. Determining where data is stored requires rich and flexible data-location representations and associated operations. Fetching small amounts of data can be done naively, one element at a time, but for large amounts of data, e.g., data used for visualization, faster

```
1       Thread is interrupted.
2       #0: MANDEL_VAL(X = <<differing COMPLEX(KIND=8) values>>, NMIN = 255, NMAX = 510)
3        at mb.hpf.f90:45,44,45,40,39
4          39               xr2 = xr*xr
5          40               xi2 = xi*xi
6          41               xi = 2*(xr*xi) + xorgi
7          42               keepgoing = n < nmax
8          43               rad2 = xr2 + xi2
9          44               xr = xr2 - xi2 + xorgr
10         45               if (keepgoing .AND. (rad2 <= 4.0)) cycle
11
12    debugger> print x
13        $1 = #<DIFFERING-VALUES
14               #0:   (-0.66200000000000003,-0.114)
15               #1:   (-0.59599999999999997,-0.113)
16               #2:   (-0.65300000000000002,-0.112)
17               #3:   (-0.93799999999999994,-0.10600000000000001)
18               #4:   (-0.56600000000000006,-0.11)
19               >
20
21    debugger> up
22        #1: hpf$hpf_fill_in_data_(TARGET = <<non-atomic= INTEGER(KIND=1), DIMENSION(1:400, 1:400)>>,
23                              W = 400, H = 400,
24                              CCR = -0.76000000000000001, CCI = -0.02, CSTEP = 0.001,
25                              NMIN = 255, NMAX = 510)
26         at mb.hpf.f90:14
27           14          forall(ix = 1:w, iy = 1:h)                                &
28
29    debugger> info address target
30        #<locative_to_hpf_section 5 peers of type INTEGER(KIND=1), DIMENSION(1:400,1:400) >
31          type           INTEGER(KIND=1), DIMENSION(1:400,1:400)
32          phys_count     5
33          addresses
34                0:          0x11fff71f0
35                1:          0x11fff7000
36                2:          0x11fff7000
37                3:          0x11fff7000
38                4:          0x11fff7000
39          arank          2
40          trank          2
41          diminfos        dlower  dupper  plower  pupper ... dist_k
42                0              1     400       1     400 ... collap
43                1              1     400       1      80 ... cyclic
44
45    debugger> info address target(100,100)
46        #<locative_in_peer in peer 4 ... >
47          type           INTEGER(KIND=1)
48          peernum        4
49          locative       #<locative_to_memory at dmem address 0x11fff8e13 of type INTEGER(KIND=1) >
```

Figure 4
Program Interrupted during Computation

methods are needed. Displaying data can usually use the techniques inherited from the underlying Fortran 90 support, but some mechanism and corresponding user interface handling is needed when replicated data has different values.

Data-Location Representations

Representing where data is stored is relatively easy to do in languages such as C and Fortran 77: the data is in a register or in a contiguous block of memory. Fortran 90 introduced assumed-shape and deferred-shape arrays,[13] where successive array elements are not necessarily adjacent in memory. HPF allows the array to be distributed so that successive array elements are not necessarily stored in a single process or address space. These lead to data that can be stored discontiguously in memory as well as in different memories.

Fortran 90 also introduced array sections, vector-valued subscripts, and field-of-array operations,[14] which further complicate the notion of where data is stored. Although evaluating an expression involving an array can be accomplished by reading the entire array and performing the operations in the debugger, this approach is inefficient, especially for a result that is sparse compared to the entire array. A standard technique is to perform address arithmetic and fetch only

```
 1    debugger> where
 2        > #0(unsync) MANDEL_VAL at mb.hpf.f90:45,44,45,40,39
 3          #1(synchr) hpf$hpf_fill_in_data_ at mb.hpf.f90:14
 4          #2(synchr) hpf_fill_in_data_ at mb.hpf.f90:1
 5          #3(scalar) mb_fill_in_data at mb.hpf.c:45
 6          #4(scalar) main at mb.c:421
 7          #5(unsync) _hpf_twinning_main_usurper at [...]/libhpf/hpf_twin.c:499,506,506,506,506
 8          #6(synchr) __start at [...]/alpha/crt0.s:361
 9    debugger> focus 1-4
10    debugger> where
11        > #0(unsync) MANDEL_VAL at mb.hpf.f90:<none>,44,45,40,39
12          #1(synchr) hpf$hpf_fill_in_data_ at mb.hpf.f90:14
13          #2(synchr) hpf_fill_in_data_ at mb.hpf.f90:1
14          #3(synchr) _hpf_non_peer_0_to_dispatch_loop at [...]/libhpf/hpf_twin.c:575
15          #4(synchr) _hpf_twinning_main_usurper at [...]/libhpf/hpf_twin.c:506
16          #5(synchr) __start at [...]/alpha/crt0.s:361
```

Figure 5
Control Flow of a Twinned Program Interrupted during Computation

```
 1    debugger> where
 2        > #0(scalar) __poll at <<unknown name>>:41
 3          #1(scalar) <<disembodied>> at <<unknown>>:459
 4          #2(scalar) _XRead at <<unknown name>>:1110
 5          #3(scalar) _XReadEvents at <<unknown name>>:950
 6          #4(scalar) XNextEvent at <<unknown name>>:37
 7          #5(scalar) HandleXInput at mb.c:58
 8          #6(scalar) main at mb.c:452
 9          #7(unsync) _hpf_twinning_main_usurper at [...]/libhpf/hpf_twin.c:499,506,506,506,506
10          #8(synchr) __start at [...]/alpha/crt0.s:361
11    debugger> focus 1-4
12    debugger> where
13        > #0(unsync) __select at <<unknown name>>:<none>,41,<none>,41,41
14          #1(unsync) TCP_MsgRead at [...]/libhpf/msgtcp.c:<none>,1057,<none>,1057,1057
15          #2(multi)
16              <none>
17              _TCP_RecvAvail at [...]/libhpf/msgtcp.c:1400
18              swtch_pri at <<unknown name>>:118
19              _TCP_RecvAvail at [...]/libhpf/msgtcp.c:1400
20              _TCP_RecvAvail at [...]/libhpf/msgtcp.c:1400
21          #3(unsync) _hpf_Recv at [...]/libhpf/msgmsg.c:<none>,434,488,434,434
22          #4(synchr) _hpf_RecvDir at [...]/libhpf/msgmsg.c:509
23          #5(synchr) _hpf_non_peer_0_to_dispatch_loop at [...]/libhpf/hpf_twin.c:563
24          #6(synchr) _hpf_twinning_main_usurper at [...]/libhpf/hpf_twin.c:506
25          #7(synchr) __start at [...]/alpha/crt0.s:361
```

Figure 6
Control Flow of a Twinned Program Interrupted While Idle in Scalar Mode

the actual data result at the end of the operation. The usual notion of an address, however, is that it describes the start of a contiguous block of memory.

Richer data-location representations are necessary. These representations can include registers and contiguous memory, but they also need to include discontiguous memory and data distributed among multiple processes. The representations should also include the results of expressions involving array sections, vector-valued subscripts, and field-of-array operations, thereby extending address arithmetic to data-location arithmetic. Aardvark defines a *locative* base class that has a virtual method to fetch the data. A variety of derived classes implement the data-location representations needed.

DIGITAL's Fortran 90 implements assumed-shape and deferred-shape arrays using descriptors that contain run-time information about the memory address of the first element, the bounds, and per-dimension inter-element spacing.[15] Aardvark models these types of arrays almost directly with a derivation of the locative class that holds the same information as the descriptor. Performing expression operations is relatively easy. An array section expression adjusts the bounds and the inter-element spacing. A field-of-array operation offsets the address to point to the compo-

nent field and changes the element type to that of the field. A vector-valued subscript expression requires additional support; the representation for each dimension can be a vector of memory offsets instead of bounds and inter-element spacing.

All arrays in HPF are qualified, explicitly or implicitly, with `ALIGN`, `TEMPLATE`, and `DISTRIBUTE` directives.[16] DIGITAL's HPF uses a superset of the Fortran 90 descriptors to encode this information. Aardvark models HPF arrays with another derivation of the locative class that holds information similar to the HPF descriptors. The most pronounced difference is that Aardvark uses a single locative to encode the descriptors from the set of processes. Aardvark knows that the local memory addresses are potentially different on each process and maintains them as a vector, but currently assumes that processor-independent information is the same on all processes and only encodes that information once.

Referring again to Figure 4, line 22 shows that the argument `TARGET` is an array, and line 29 is a request for information about the location of its data. (See also Figure 3 for the full source, including the declaration and distribution of `TARGET`.) Figure 4, line 32 shows that there are five processes, and lines 34 through 38 show the base address within each process. The addresses for processes 1 through 4 happen to be the same, but the address for process 0 is different. Lines 39 and 40 show that the rank of the array (`arank`) and the rank of the template (`trank`) are both 2. Lines 42 and 43 show the dimension information for the array. The declared bounds are `1:400,1:400`, but the local physical bounds are `1:400,1:80` and the distribution is `(*,CYCLIC)`. This is all accurate; distributing the second dimension on five processes causes the local physical size for that dimension (80) to be one-fifth the declared bound (400).

Performing expression operations on HPF-based locatives is more involved than for Fortran 90. Processing a scalar subscript not only offsets the base memory address but also restricts the set of processors determined by the dimension's distribution information. Processing a subscript triplet, e.g., `from:to:stride`, involves adjusting the declared bounds and the alignment; it does not adjust the template or the physical layout. As in Fortran 90, processing a vector-valued subscript in HPF requires the locative to represent the effect of the vector. For HPF, the representation is pairs of memory offsets and processor set restrictions. Processing a field-of-array operation adjusts the element type and offsets each memory address.

When selecting a single array element by providing scalar subscripts, another type of locative is useful. This locative describes on which process the data is stored and a locative relative to that selected process. For example, line 45 of Figure 4 requests the location information of a single array element. The result shows that it is on process 4 at the memory address indicated by the contained locative.

Fetching HPF Data

As just mentioned, locatives provide a method to fetch the data described by the locative. For a locative that describes a single distributed array element (e.g., Figure 4, lines 45 through 49), the method extracts the appropriate physical thread from the logical thread and uses the contained locative to fetch the data relative to the extracted physical thread. For a locative that describes an HPF array, Aardvark currently iterates over the valid subscript space, determines the physical process number and memory offset for each element, and fetches the element from the selected physical process. For small numbers of elements, on the order of a few dozen, this technique has acceptable performance. For large numbers of elements, e.g., for visualization or reduction operations, the cumulative processing and communication delay to retrieve each individual element is unacceptable. This performance issue also exists for locatives that describe discontiguous Fortran 90 arrays. The threshold is higher because there is no computation to determine the process for an element, and the process is usually local rather than remote, eliminating communication delays.

The primary bottleneck is issuing many small data retrieval requests to each (remote) process. This involves many communication delays and many delays related to retrieving each element. What is needed is to issue a smaller number of larger requests. The smaller number reduces the number of communication transactions and associated delays. Larger requests allow analysis of a request to make more efficient use of the operating system's mechanisms to access process memory. For example, a sufficiently dense request can read the encompassing memory in a single call to the operating system and then extract the desired elements once the data is within the debugger.

Although not implemented, the best solution, in my opinion, is to provide a "read (multidimensional) memory section" method on a process in addition to the common "read (contiguous) memory" method. If the process is remote, as it usually is with HPF, the method would be forwarded to a remote debug server controlling the remote process. The implementation of the method that interacts with the operating system would know the trade-offs to determine how to analyze the request for maximum efficiency.

Converting a locative describing a Fortran 90 array section to a "read memory section" method should be easy: they represent nearly the same thing. For a locative that describes a distributed HPF array, Aardvark would need to build (physical) memory section descriptions for each physical process. This can be done by iterating over the physical processes and building the memory section for each process. It is

also possible to build the memory sections for all the processes during a single pass through the locative, but the performance gains may not be large enough to warrant the added complexity.

Differing Values

Using HPF to distribute an array often partitions its elements among the processes. Scalars, however, are generally replicated and may be expected to have the same value in each process. There are cases, though, where seemingly replicated scalars may not have the same value. DO loops that do not require data to be communicated between processes do not have synchronization points and can become out of phase, resulting in their indexes and other privatized variables having different values. Functions called within a FORALL construct often run independently of each other, causing the arguments and local variables in one process to be different from those in another. A debugger should be aware that values might differ and adjust the presentation of such values accordingly.

Aardvark's approach is to define a new kind of value object called *differing values* to represent a value from a semantically single source that does not have the same value from all its actual sources. A user interface can detect this kind of value and display it in different ways, for example, based on context and/or the size of the data.

Referring again to Figure 4, the program was interrupted while each process was executing the function MANDEL_VAL called within a FORALL. Line 2 shows that the argument X was determined to have differing values. This user interface does not show all the values at this point; a large number of values could distract the user from the current objective of discovering where the process stopped. Instead, it shows an indication that the values are different along with the type of the variable. Notice that the other two arguments, NMIN and NMAX, are presented as integers; they have the same value in all processes. Line 12 requests to see the value of X. Line 13 again shows that the values are different, and lines 14 through 18 show the process number and the value from the process.

To build a differing values object, Aardvark reads the values for a replicated scalar from each process. If all the values are bit-wise equal, they are considered to be the same and a standard (single) value object is returned. Otherwise, a differing values object is constructed from the several values. For numeric data, this approach seems reasonable. If the value of a scalar integer variable INTVAR is 4 on all the processes, then 4 is a reasonable (single) value for INTVAR. If the value of INTVAR is 4 on some processors and 5 on others, no single value is reasonable. For nonnumeric data and pointers, there is the possibility of false positives and false negatives. The ideal for user-defined types is to compare the fields recursively. Pointers that are semantically the same can point to targets located at different memory addresses for unrelated reasons, leading to different memory address values and therefore a false positive. To correctly dereference the pointers, though, Aardvark needs the different memory address values. In short, it is reasonable to test numeric data and create a single value object or a differing values object, and it appears reasonable to do the same for nonnumeric data, despite the possibility of a technically false kind of value object.

Currently, differing values do not participate in arithmetic. That is, the expression INTVAR.LT.5 is valid if INTVAR is a single value but causes an error to be signaled if INTVAR is a differing value. Many cases could be made to work, but some cases defy resolution. In the INTVAR.LT.5 case, if all values of INTVAR are less than 5 or all are greater than or equal to 5, then it is reasonable to collapse the result into a single value, .TRUE. or .FALSE., respectively. If some values are less than 5 and some are not, it also seems reasonable to create a differing values object that holds the differing results. What if INTVAR.LT.5 is used as the condition of a breakpoint and some values of INTVAR are less than 5 and some are not? The breakpoint should probably cause the process (and all the physical processes) to remain stopped. It is unclear whether arithmetic on differing values would be useful to users or if it would lead to more confusion than it would clear up.

Unmet Challenges

HPF presents a variety of challenges that Aardvark does not yet address. Some of these challenges are not in common practice, giving them low priority. Some are recent with HPF Version 2.0 and are being used with increasing frequency. Some of the challenges, for example, a debugger-initiated call of an HPF procedure, are tedious to address correctly.

Mapped Scalars

It is possible to distribute a scalar so that the scalar is not fully replicated.[17] The compiler would need to emit sufficient debugging information, which would probably be a virtual array descriptor with an array rank of 0 and a nonzero template rank. Aardvark would probably model it using its existing locative for HPF arrays, also with an array rank of 0 and appropriate template information.

Replicated Arrays

Unless otherwise specified, DIGITAL's HPF compiler replicates arrays. It is possible to replicate arrays explicitly and to align arrays (and scalars) so that they are partially replicated. Currently, Aardvark does not detect a replicated array, despite the symbol table or run-time descriptor indicating that it is replicated. As a result, Aardvark determines a single process from which to fetch each array element. For fully replicated

arrays, Aardvark should read the array from each process and process them with the differing values algorithms. Correctly processing arrays that are partially replicated is not as easy as processing unreplicated or fully replicated arrays. If the odd columns are on processes 0 and 1, while the even columns are on processes 2 and 3, no single process contains the entire array. The differing values object would need to be extended to index the values by a processor set rather than a single process.

Update of Distributed and Replicated Objects

Aardvark currently supports limited modification of data. It supports updating a scalar object (scalar variable or single array element) with a scalar value, even if the object is distributed or replicated. Even this can be incorrect at times. Assigning a scalar value to a replicated object sets each copy, which is undesirable if the object has differing values. Assigning a value that is a differing values object is not supported. More importantly (and more subtly), Aardvark is not aware of shadow or halo copies of data that are stored in multiple processes, so updating a distributed object updates only the primary location.

Distributed Array Pointers

HPF Version 2.0 allows array pointers in user-defined types to be distributed and allows fully replicated arrays of such types. For example, in

```
type utype
   integer, pointer :: compptr(:)
   !hpf$ distribute compptr(block)
end type

type (utype) :: scalar, array(20)
```

the component field `compptr` is a distributed array pointer. Aardvark does not currently process the array descriptor(s) for `scalar%compptr` at the right place and as a result does not recognize the expression as an array. As mentioned earlier, Aardvark reads a replicated array element from a single process. To process `array(1)%compptr`, all the descriptors are needed, e.g., for the base memory addresses in the physical processes. The use of this relatively new construct is growing rapidly, elevating the importance of being supported by debuggers.

Ensuring a Consistent View

A program can have its physical threads stop at the same place but be in different iterations of a loop. Aardvark mistakenly presents this state as synchronized and presents data as if it were consistent. This is what is happening in Figures 4 and 5; `hpf$hpf_fill_in_data` (frame #1) is in different iterations of the `FORALL`. With compiler assistance, it is possible to annotate each thread's location with iteration counts in addition to traditional line numbers.[18] The resulting set of locations can be compared to a location in the conceptually serial program to determine which threads have already reached (and perhaps passed) the serial location and which have not yet reached it. A debugger could automatically, or under user control, advance each thread to a consistent serial location. For now, Aardvark's differing values mechanism is the clue to the user that program state might not be consistent.

Calling an HPF Procedure

Having a debugger initiate a call to a Fortran 90 procedure is difficult in the general case. One difficulty is that copy-in/copy-out (making a temporary copy of array arguments and copying the temporary back to its origin after the call returns) may be necessary. HPF adds two more difficulties. First, the data may need to be redistributed, which amounts to a distributed copy-in/copy-out and entails a lot of tedious (but hopefully straightforward) bookkeeping. Second, an HPF thread's state is much more complex than a collection of physical thread states. When a debugger initiates a uniprocessor procedure call, it generally saves the registers, sets up the registers and stack according to the calling convention, lets the process run until the call returns, extracts the result, and finally restores the registers. The registers are generally the state that is preserved across a debugger-initiated procedure call. For HPF, and in general for other paradigms that use message passing, it may be necessary to preserve the run-time state of the messaging subsystem in each process. This preservation probably amounts to making uniprocessor calls to messaging-supplied save/restore entry points, allowing the messaging subsystem to define what its state is and how it should be saved and restored. Although logical entities would be used to coordinate the physical details, this is a lot of work and has not been prototyped.

Related Work

DIGITAL's representative to the first meeting of the HPF User Group reported a general lament among users about the lack of debugger support.[19,20] Browsing the World Wide Web reveals little on the topic of HPF debugging, although some efforts have provided various degrees of sophistication.

Multiple Serial Debuggers

A simplistic approach to debugging support is to start a traditional serial debugger on each component process, perhaps providing a separate window for each and providing some command broadcast capability. Although this approach provides basic debugging, it does not address any of the interesting challenges of HPF debugging.

Prism

The Prism debugger (versions dating from 1992), formerly from Thinking Machines Corporation, provides debugging support for CM Fortran.[21,22] The run-time model of CM Fortran is essentially single instruction, multiple data (SIMD), which considerably simplifies managing the program. The program gets compiled into an executable that broadcasts macroinstructions to the parallel machine, even on the CM-5 synchronized multiple instruction, multiple data (MIMD) machine. Prism primarily debugs the single program doing the broadcasting. Therefore, operations such as starting, stopping, and setting breakpoints can use the traditional uniprocessor debugging techniques. Prism is aware of distributed data. When visualizing a distributed array, however, it presents each process's local portion and conceptually augments the rank of the array to include a process axis. For example, a two-dimensional 400 x 400 array distributed `(*,CYCLIC)` on five processes is presented as a 400 x 80 x 5 array. For explicit message sending programs, Prism controls the target processes and provides a "where graph," which has some of the visual cues that Aardvark's logical frames provide.

TotalView

Recent (1997) versions of the TotalView debugger, from Dolphin Interconnect Solutions, Inc., provide some support for the HPF compiler from The Portland Group, Inc.[23,24] TotalView provides "process groups," which are treated more like sets for set-wide operations than like a synthesis into a single logical entity. As a result, no unified view of the call stacks exists. TotalView can "dive" into a distributed HPF array and present it as a single array in terms of the original source. Distributed data is not currently integrated into the expression system, however, so a conditional breakpoint such as `A(3,4).LT.5` does not work. TotalView is being actively developed; future versions will likely provide more complete support for HPF.

Applicability to Other Areas

Many of the techniques that Aardvark incorporates can apply to other areas, including the single program, multiple data (SPMD) paradigm, debugging optimized code, and interpreted languages.

Single Program, Multiple Data

Logical entities can be used to manage and examine programs that use the SPMD paradigm. This is true for process-level SPMD, which is commonly used with explicit message sending such as MPI,[5,6] and for thread-level SPMD such as directed decomposition.[25-27] Aardvark's twinning algorithms can be used in both cases. Process-level SPMD is similar to DIGITAL's HPF; the equivalent of twinning requires a stylistic way of coding and declaring a dispatch loop. Thread-level SPMD usually has a pool of threads waiting in a dispatch loop, requiring Aardvark to know some mechanics of the run-time support.

The differing values mechanism can apply to data in SPMD paradigms. DIGITAL's recent introduction of Thread Local Storage (TLS),[28] modeled on the Thread Local Storage facility of Microsoft Visual C++[29] with similarities to `TASKCOMMON` of Cray Fortran,[30] provides another source of the same variable having potentially differing values in different thread contexts.

Debugging Optimized Code

Aardvark's flexible locative subsystem and its awareness of nonsingular values (i.e., differing values) can be the basis for "split-lifetime variables." In optimized code, a variable can have several simultaneous lifetimes (e.g., the result of loop unrolling) or no active lifetime (e.g., between a usage and the next assignment). New derivations of the locative class can describe the multiple homes or the nonexistent home of a variable. Fetching by means of such a locative creates new kinds of values that hold all the values or an indication that there is no value. User interfaces become aware of these new kinds of values in ways similar to their awareness of differing values.

Aardvark's method of asking a thread for a single-stepping run reason and empowering the reason to accomplish its mission can be the basis for single stepping optimized code. Optimized code generally interleaves instructions from different source lines, rendering the standard "execute instructions until the source line number changes" method of single stepping useless. If instead the compiler emits information about the semantic events of a source line, Aardvark can construct a single-stepping run reason based on semantic events rather than line numbers. Single stepping an optimized HPF program immediately reaps the benefits since logical stepping is built on physical stepping.

Interpreted Languages

Logical entities can be used to support debugging interpreted languages such as Java[31] and Tcl.[32] In this case, the physical process is the operating system's process (the Java Virtual Machine or the Tcl interpreter), and the logical process is the user-level view of the program. A logical stack frame encodes a procedure call of the source language. This is accomplished by examining virtual stack information in physical memory and/or by examining physical stack frames, depending on how the interpreter is implemented. Variable lookup within the context of a logical frame would use the interpreter-managed symbol tables rather than the symbol tables of the physical process.

Summary

HPF presents a variety of challenges to a debugger, including controlling the program, examining its call stack, and examining its data, and user interface implications in each area. The concept of logical entities can be used to manage much of the control complexity, and a rich data-location model can manage HPF arrays and expressions involving arrays. Many of these ideas can apply to other debugging situations. On the surface, debugging HPF can appear to be a daunting task. Aardvark breaks down the task into pieces and attacks them using powerful extensions to familiar ideas.

Acknowledgments

I am grateful to Ed Benson and Jonathan Harris for their unwavering support of my work on Aardvark. I also thank Jonathan's HPF compiler team and Ed's Parallel Software Environment run-time team for providing the compiler and run-time products that allowed me to test my ideas.

References and Notes

1. *Programming Language Fortran 90*. ANSI X3.198-1992 (New York, N.Y.: American National Standards Institute, 1992).

2. J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener, *Fortran 90 Handbook* (New York, N.Y.: McGraw-Hill, 1992).

3. High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*. This specification is available by anonymous ftp from softlib.rice.edu in the directory pub/HPF. Version 2.0 is the file hpf-v20.ps.gz.

4. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel, *The High Performance Fortran Handbook* (Cambridge, Mass.: MIT Press, 1994).

5. MPI Forum, "MPI-2: Extensions to the Message-Passing Interface," available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html or via the Forum's documentation page http://www.mpi-forum.org/docs/docs.html.

6. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference* (Cambridge, Mass.: MIT Press, 1995).

7. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI* (Cambridge, Mass.: MIT Press, 1994).

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

9. E. Benson et al., "Design of Digital's Parallel Software Environment," *Digital Technical Journal*, vol. 7, no. 3 (1995): 24–38.

10. It is possible to always use logical entities, but sometimes it is easier to work with the building blocks when additional structure might be cumbersome. In a similar vein, Fortran could eliminate scalars in favor of arrays of rank 0, but Fortran chooses to retain scalars because of their ease of use.

11. In this policy, *nasty* and *nice* are generic names for categories. Which particular stop reasons fall into which category is a separate design question. Once the category is determined, the policy presented can be performed.

12. Debuggers often build a (physical) call stack from innermost frame to outermost frame, interleaving construction with presentation. The interleaving gives the appearance of progress even if there are occasional delays between frames. The total time required to reach the outermost frames of each physical thread, which must occur before construction of a logical call stack can begin and before any presentation is possible, could be noticeable to the user.

13. An assumed shape array is a procedure array argument in which each dimension optionally specifies the lower bound and does not specify the upper bound. For example,

```
REAL :: ARRAY_ARG_2D(:,4:)
```

A deferred shape array has either the `ALLOCATABLE` or `POINTER` attribute, specifies neither the lower nor upper bound, and often contains local procedure variables or module data. For example,

```
REAL, ALLOCATABLE :: ALLOC_1D(:)
REAL, POINTER :: PTR_3D(:,:,:)
```

14. An array section occurs when some subscript specifies more than one element. This can be done with a subscript-triplet, which optionally specifies the lower and upper extents and a stride, and/or with an integer vector, for example,

```
ARRAY_3D(ROW1:ROWN , COL1::COL_STRIDE , PLANE_VEC)
```

A field-of-array operation specifies the array formed by a field of each structure element of an array, for example,

```
TYPE (TREE) :: TREES(NTRESS)
REAL :: TREE_HEIGHTS(NTREES)
TREE_HEIGHTS = TREES%HEIGHT
```

In general, each of these specifies discontiguous memory.

15. "DEC Fortran 90 Descriptor Format," *DEC Fortran 90 User Manual* (Maynard, Mass.: Digital Equipment Corporation, June 1994).

16. The HPF array descriptor for the variable `ARRAY` in the HPF fragment

```
!HPF$ TEMPLATE T(NROWS,NCOLS)
!HPF$ DISTRIBUTE T(CYCLIC,BLOCK)
REAL :: ARRAY(NCOLS/2,NROWS)
!HPF$ ALIGN ARRAY(I,J) WITH T(J,I*2-1)
```

contains components corresponding to each of the ALIGN, TEMPLATE, and DISTRIBUTE directives. Often an array is distributed directly, causing the ALIGN and TEMPLATE directives to be implicit, for example,

```
REAL :: MATRIX(NROWS,NCOLS)
!HPF$ DISTRIBUTE MATRIX(BLOCK,BLOCK)
```

17. The variable SCALAR in

```
!HPF$ TEMPLATE T(4,4)
!HPF$ DISTRIBUTE T(CYCLIC,CYCLIC)
!HPF$ ALIGN SCALAR WITH T(*,2)
```

is partially replicated and will be stored on the same processors that the logical second column of the template T is stored.

18. R. Cohn, *Source-Level Debugging of Automatically Parallelized Programs*, Ph.D. Thesis, Carnegie Mellon University (October 1992).

19. HPF User Group, February 23–26, 1997, Santa Fe, New Mexico. Information about the meeting is available at http://www.lanl.gov/HPF/.

20. In a trip report, DIGITAL's representative Carl Offner reported the following: "Many people complained about the lack of good debugging support for HPF. Those who had seen our [Aardvark-based] debugger liked it a lot.... [An industrial HPF user] complained emphatically about the lack of good debugging support.... [Another industrial HPF user's] biggest concern is the lack of good debugging facilities."

21. *Prism User's Guide* (Cambridge, Mass.: Thinking Machines Corporation, 1992).

22. *CM Fortran Programming Guide* (Cambridge, Mass.: Thinking Machines Corporation, 1992).

23. *TotalView: User's Guide* (Dolphin Interconnect Solutions, Inc., 1997). This guide is available via anonymous ftp from ftp.dolphinics.com in the totalview/DOCUMENTATION directory. At the time of writing the file is TV-3.7.5-USERS-MANUAL.ps.Z.

24. *PGHPF User's Guide* (Wilsonville, Ore.: The Portland Group, Inc., 1997).

25. *KAP Fortran 90 for Digital UNIX* (Maynard, Mass.: Digital Equipment Corporation, October 1995).

26. "Fine-Tuning Power Fortran," *MIPSpro POWER Fortran 77 Programmer's Guide* (Mountain View, Calif.: Silicon Graphics Inc, 1994–1996).

27. "Compilation Control Statements and Compiler Directives," *DEC Fortran 90 User Manual* (Maynard, Mass.: Digital Equipment Corporation, forthcoming in 1998).

28. *Release Notes for [Digital UNIX] Version V4.0D* (Maynard, Mass.: Digital Equipment Corporation, 1997).

29. "The Thread Attribute," in *Microsoft Visual C++: C++ Language Reference (Version 2.0)* (Redmond, Wash.: Microsoft Press, 1994): 389–391.

30. *CF90 Commands and Directives Reference Manual* (Eagan, Minn.: Cray Research, Inc., 1993, 1997).

31. J. Gosling and H. McGilton, *The Java Language Environment: A White Paper* (May 1996). This paper is available at http://www.javasoft.com/docs/white/langenv/ or via anonymous ftp from ftp.javasoft.com in the directory docs/papers, for example, the file whitepaper.ps.tar.Z.

32. J. Ousterhout, *Tcl and the Tk Toolkit* (Reading, Mass: Addison-Wesley, 1994).

Biography

David C. P. LaFrance-Linden
David LaFrance-Linden is a principal software engineer in DIGITAL's High Performance Technical Computing group. Since joining DIGITAL in 1991, he has worked on tools for parallel processing, including the HPF-capable debugger described in this paper. He has also contributed to the implementation of the Parallel Software Environment and to the compile-time performance of the HPF compiler. Prior to joining DIGITAL, David worked at Symbolics, Inc. on front-end support, networks, operating system software, performance, and CPU architecture. He received a B.S. in mathematics from M.I.T. in 1982.

digital

ISSN 0898-901X

Printed in U.S.A. EC-P8826-20/98 01 19 24.0 Copyright © Digital Equipment Corporation