DECnet for OpenVMS AXP: A Case History

1 Abstract

The DECnet for OpenVMS AXP networking software facilitates the integration of OpenVMS AXP systems into existing DECnet computing environments. This new software product supports application migration by providing the following networking capabilities: support of compatible libraries, consistent application programming interfaces, and the assurance of a common semantic operation with the OpenVMS VAX system. The team implemented a phased porting process and executed the project cooperatively. The effort resulted in a solid knowledge base with which to approach future porting undertakings. Using common code where possible and avoiding architecture-specific code were lessons learned during the project.

The DECnet for OpenVMS AXP networking software product plays an important role in the integration of OpenVMS AXP systems into existing DECnet computing environments. The availability of DECnet software on the Alpha AXP hardware platform facilitates application migration. The networking capabilities needed to support this migration activity include support of compatible libraries, consistent application programming interfaces (APIs), and the assurance of a common semantic operation with the OpenVMS VAX system. The network features such as network file transfer, remote file access, remote login, downline load, and local and remote network management allow the OpenVMS AXP system to participate fully in a DECnet network.

The purpose of this paper is to describe the process of porting the DECnet-VAX product to the OpenVMS AXP operating system. The DECnet-VAX product consists of networking software written in the MACRO-32 and BLISS-32 programming languages. The software contains privileged system code, device drivers, and user-mode utilities.

This paper is divided into two major sections. The first section presents an overview of the project, including discussions about the DECnet features supported in the OpenVMS AXP operating system, the project schedule, and the major DECnet for OpenVMS AXP components. The second major section details the process of porting DECnet-VAX software to the OpenVMS AXP operating system, including testing and debugging. This section provides information on nonportable coding practices and identifies specific problem areas. It concludes with a summary of the lessons learned during the course of the project.

2 Project Overview

In addition to presenting the DECnet for OpenVMS AXP features, this section details how we derived a project schedule and gives an overview of the software components.

Software Code Base

Prior to the formation of a team to port a DECnet product from VAX to the Alpha AXP architecture, the DECnet-VAX development group completed a feasibility study of porting DECnet-VAX Phase IV to the Alpha AXP architecture. This effort was necessary because the DECnet-VAX software was not designed with porting in mind. The study concluded that it would take four engineers twelve months (i.e., 48 person-months) to port DECnet-VAX to the OpenVMS AXP operating system. After examining the proposal and investigating the alternatives, we decided that the best approach would be to start by porting DECnet-VAX V5-4.3, a Digital Network Architecture (DNA) Phase IV implementation.[1] One of the most important factors in making this decision was that this software version was in external field test and was nearly ready for shipment to customers. Another consideration was that some very important fixes had been made in that release, and we wanted to offer our customers the highest quality possible in the first version of DECnet for OpenVMS AXP software. Since that time, we have continued to improve our DECnet software for the OpenVMS AXP operating system and have recently incorporated some fixes from DECnet for OpenVMS VAX V5.5-2.

DECnet for OpenVMS AXP Features

The first release of the DECnet for OpenVMS AXP networking product is packaged with the OpenVMS AXP operating system. The initial offering includes the support of DECnet Phase IV protocols running over Ethernet or fiber distributed data interface (FDDI) local area networks. This release supports distributed task-to-task communications using the same set of documented programming interfaces supported in the DECnet-VAX environment. At this time, DECnet for OpenVMS AXP software does not support wide area communications devices and host-based routing. Future releases of DECnet for OpenVMS AXP may include symmetric multiprocessor (SMP) and cluster alias support.

Project Schedule

The DECnet for OpenVMS AXP project schedule was primarily driven by the overall OpenVMS AXP operating system product schedule, with the DECnet component scheduled for delivery in November 1991. The DECnet-VAX porting project officially began in early January 1991, after the code base was selected.

Porting Estimates. After analyzing the work required to achieve the port, we developed general porting guidelines and estimates based on a number of factors, including the language the software was written in, the amount of software to port, and the number of software component modules. We then combined these estimates to determine an overall project schedule. Table 1 presents the guidelines we used for the porting estimates.

We used two methods to estimate the amount of work required to complete the port. The Module Size Method takes into account the number of lines of code per software module. The Module Count Method uses the number of modules per software component to determine the workload. Both methods take into consideration the programming language used in each module. Table 2 presents details of the component module count and sizes. We further categorized the software being ported into three groups: privileged code, device driver, and user-mode utility. The software type was used to estimate the amount of time needed for linking. In general, we allocated more time for privileged code and device drivers.

The estimates were used to derive a first-pass schedule and to determine resource allocation. A number of other factors affected the final schedule. A major factor that we could not quickly estimate was the portability of the software. The software techniques encountered and described in this paper such as coroutines, up-level stack references, and condition code usage had a direct impact on the schedule. Also, during the first three months of the project, significant time was spent learning how to port code. During this learning period, we developed the skills, knowledge, and techniques used throughout the remainder of our porting work.

Once we established the estimation metrics, the data was compiled and time estimates calculated for each component. Tables 3 and 4 show the average amount of time required to port each DECnet for OpenVMS AXP component.

Based on these calculations, we estimated that it would take 13 person-months just to port the DECnet-VAX software. We then used project management software to plan the schedule. The schedule shown in Table 5 indicated that it would take 48 person-months to meet the OpenVMS AXP scheduled completion date of November 22, 1991. We made our first network connection on July 25, 1991, 20 person-months into the project. Although much work remained, we were well ahead of the November target date.

Since we were ahead of schedule, we assisted in the porting of other components, including RTPAD, CTDRIVER, RTTDRIVER, and REMACP, all discussed later in the paper. In addition, we were able to add support for FDDI.

Milestones. The OpenVMS AXP project schedule consisted of a series of functional internal base levels numbered one to five. In terms of the whole OpenVMS AXP project schedule, DECnet for OpenVMS AXP was targeted for base level five. However, it was highly desirable to provide file transfer and remote login capability over DECnet as early as possible. The project team worked closely with the OpenVMS AXP group to deliver this support prior to base level four.

Common Code

One of the most important decisions that helped us deliver our software ahead of schedule was building common code for the VAX and Alpha AXP systems. During the course of porting code, we discovered two advantages to building common code. The first was having the ability to generate VAX and Alpha AXP images from a single set of source code. The second was being able to debug our ported changes in a stable OpenVMS VAX environment. We accomplished this by rewriting code that required change so that it worked on both platforms. We made architecture-specific code conditional on the platform on which it would execute. Our long-term goal is to incorporate common code into future DECnet for OpenVMS products.

DECnet for OpenVMS AXP Components

This section describes the major DECnet for OpenVMS AXP components and lists the porting issues relevant to each.[2] Figure 1 shows the interconnection of the various components of the DECnet for OpenVMS AXP software. Detailed information for each porting issue is further discussed in this paper under the Porting Issues heading.

NETDRIVER. NETDRIVER is a pseudo device driver, i.e., a device driver that does not directly control any hardware devices. It implements the routing, end communication, and session control layers of the Phase IV version of DNA.[1]

The queue I/O request ($QIO) system service is the interface into the session control layer. The NETDRIVER routing layer communicates with other device drivers that implement the data link layer of DNA. NETDRIVER communicates with NETACP (another component discussed later in this section) to perform network management functions, to report state and network topology changes, and to perform operations that require process context.

NETDRIVER is written in MACRO-32 code and presented us with many porting issues, including device driver changes, coroutines, memory management changes, page size dependencies, atomicity and granularity problems, OpenVMS AXP operating system data structure changes, unaligned references, and up-level stack references.

MOM. The maintenance operations module (MOM) image processes service operations defined by the maintenance operation protocol (MOP). One such service operation is downline loading remote systems. MOM uses NDDRIVER (described in the next subsection) to communicate with the remote system over a DECnet circuit. MOM communicates with NETACP to gather information about nodes requesting to be downline loaded. NETACP creates a process running the MOM image when a request for a service operation is received on a circuit enabled to perform service operations.

MOM is written primarily in BLISS-32 code. Porting issues included removing dependencies on the format of a VAX argument list, condition handling changes, and Alpha AXP image header changes.

NDDRIVER. The pseudo device driver NDDRIVER implements an interface that allows MOM to use a DECnet circuit to perform service operations using DNA MOP. The MOM image uses the $QIO system service interface to send MOP messages to and receive MOP messages from NDDRIVER, which then communicates with the data link device drivers to transmit and receive these messages. NDDRIVER communicates with NETACP to perform tasks that require process context and to receive notification of state changes to circuits enabled for service operations.

NDDRIVER is written in MACRO-32 code. Porting issues included changes to device drivers, memory management, and OpenVMS AXP operating system data structures, as well as page size dependencies.

CTDRIVER, RTTDRIVER, and REMACP. CTDRIVER is a pseudo device driver for remote terminals using the DNA command terminal (CTERM) protocol. CTDRIVER and RTTDRIVER perform similar functions with the exception that RTTDRIVER is used for interoperability with older implementations of remote terminal support. REMACP is an ancillary control process (ACP) that receives incoming requests for remote terminal support. After REMACP establishes a connection with the remote node, either CTDRIVER or RTTDRIVER communicates directly with NETDRIVER to send and receive remote terminal protocol messages.

CTDRIVER, RTTDRIVER, and REMACP are written in MACRO-32 code and presented the following porting issues: device driver changes, unaligned references, OpenVMS AXP operating system data structure changes, and for REMACP, changes in the interface with CTDRIVER.

NETACP. NETACP runs as an ACP that assists NETDRIVER in performing network operations that require process context. Examples include creating processes for incoming logical links and assigning channels to data link devices. NETDRIVER and NETACP also work together to maintain information about the state of the network. Another major function performed by NETACP is the management of the network configuration parameters residing in virtual memory.

NETACP is written in MACRO-32 code. Porting issues included coroutines, usage of processor status longword (PSL) condition codes, memory management changes, page size dependencies, atomicity and granularity problems, OpenVMS AXP operating system data structure changes, and unaligned references. In addition, the use of "poor programmer's lockdown," a method of locking pages into a working set, required modification.

NETSERVER. The NETSERVER image is run by server processes created to handle incoming logical link requests. NETSERVER invokes the image or command procedure associated with the network object specified by the incoming logical link. To avoid the overhead of process creation, a server process can be reused after the logical link it was servicing is terminated. Idle server processes register themselves with NETACP so that they may be reused for another logical link.

NETSERVER is written in BLISS-32 code. The only porting change necessary was the addition of the BLISS VOLATILE attribute to several data declarations.

NCP. The network control program (NCP) is the user interface for network management. NCP communicates with other network management components using the network information and control exchange (NICE) protocol. NCP can be used to manage the local node as well as remote nodes. When managing the local node, NCP exchanges NICE protocol messages with the NMLSHR shareable image. For remote management, NCP creates a logical link to the network management listener (NML) object on the remote node and exchanges NICE protocol messages over this logical link.

NCP consists primarily of BLISS-32 modules. The major porting issue associated with NCP was changing the code to use LIB$TABLE_PARSE rather than LIB$TPARSE.

NMLSHR. NMLSHR is a shareable image that processes NICE protocol network management messages on an OpenVMS system. NMLSHR decodes NICE messages received as input and performs the requested management operation. NMLSHR builds NICE protocol messages as a response to requests asking for network management information to be returned. NCP and NML both link with the NMLSHR image to call the routines that process the NICE protocol messages.

NMLSHR is written in BLISS-32 and MACRO-32. Porting issues included dependencies on the format of a VAX argument list and changes required to link shareable images.

NML. The network management listener (NML) image is run when a remote node requests a connection to the NML object to perform remote network management operations. NML sends NICE protocol messages to and receives them from the remote node. NML passes NICE protocol messages received from the remote node to NMLSHR for decoding and receives messages from NMLSHR to send to the remote node.

NML is written in BLISS-32 code. The only porting change made to NML code was to add the BLISS VOLATILE attribute to one data declaration.

EVL. The event logger (EVL) receives event messages from the various DNA layers. EVL can also act as an event sink for messages generated at a remote node. EVL is started by NETACP and declares itself as a network object so that remote nodes can connect to the EVL object and send event messages. EVL can log events to a file in binary form or format the messages into something readable by a network manager.

EVL is written in BLISS-32 code. Porting issues included adding the BLISS VOLATILE attribute to some data structure definitions and aligning data structure fields on natural boundaries.

DTS and DTR. The DECnet test sender (DTS) and the DECnet test receiver (DTR) are cooperating programs that can be used to test the network connection between two nodes. DTS runs on the local node and communicates with DTR on the remote node. DTS and DTR can be used to test the throughput and reliability of a line over which DECnet is running.

DTS and DTR are written primarily in MACRO-32 code. The two major porting issues associated with DTS and DTR were changing the code to use LIB$TABLE_PARSE rather than LIB$TPARSE and adding some BLISS-32 code to support floating-point operations.

RTPAD. RTPAD provides the connection between a local terminal and the remote terminal services of a remote node. RTPAD is invoked as the result of executing the SET HOST command of the Digital Command Language (DCL). RTPAD communicates with REMACP and CTDRIVER or RTTDRIVER on the remote system to provide remote terminal support. RTPAD accepts input from the local terminal (which could be another remote terminal) and sends this data over the network to the remote node. Output from the remote node is received by RTPAD and displayed on the local terminal.

RTPAD is written in MACRO-32 code. Porting issues included unaligned references and aligning data structure fields on natural boundaries.

NICONFIG. NICONFIG is the Ethernet configurator that listens to the MOP system identification messages broadcast on Ethernet circuits and maintains a database of configuration information for all systems heard. NCP is used to manage and display the information maintained by NICONFIG. NICONFIG runs as a process created by NMLSHR and communicates with NMLSHR over a DECnet logical link using the NICE protocol.

NICONFIG is written in BLISS-32 code. The only porting change was to remove the module switch LANGUAGE.

HLD. The host loader (HLD) communicates with the DECnet-RSX satellite loader to downline load tasks to an RSX-11S node. HLD is written in MACRO-32 code. The only porting change was to update the structure definition language used to create one data structure.

MIRROR. The loopback mirror participates in network services protocol and routing layer loopback testing. MIRROR is written in MACRO-32 code. No porting changes were required though changes were made to the link procedure.

3 DECnet-VAX Port to the OpenVMS AXP Operating System

This section discusses the development environment, process, and issues related to porting the DECnet-VAX product to the OpenVMS operating system.

DECnet for OpenVMS AXP Development Environment

DECnet for OpenVMS AXP is built with and integrated into the OpenVMS AXP operating system. Many changes were being made to system data structures that directly affected the DECnet software. These changes required the DECnet for OpenVMS AXP software to be built with and tested on many interim operating system base levels before the combined OpenVMS AXP operating system and DECnet for OpenVMS AXP kit was shipped for layered product development.

Because the development tools changed throughout the project, we used the same tools to port the DECnet-VAX software as were used to develop the operating system base levels. When we copied portions of an OpenVMS AXP base level, we also copied the tool directories associated with the system build. We used cross compilers for MACRO-32 and BLISS-32, which allowed us to develop Alpha AXP software on an OpenVMS VAX system.[3] In addition, we used the OpenVMS AXP linker, librarian, and system dump analyzer (SDA) cross tools on the VAX system.[4,5] We also used the OpenVMS AXP debugging tools Delta and XDelta on the Alpha AXP prototype hardware.[6]

Initial DECnet for OpenVMS AXP testing was accomplished on a VAX system. Such testing was possible because we designed a majority of the DECnet for OpenVMS AXP code to run on both VAX and Alpha AXP hardware platforms.

The Alpha AXP prototype system used for testing utilized a shared disk that contained the OpenVMS AXP operating system images. The images and test procedures were copied onto the disk from a VAX system. Each time new DECnet for OpenVMS AXP images or test procedures had to be added to the shared disk during a test or debug session, the Alpha AXP test system had to be stopped, the disk mounted on the VAX system, images copied, the disk dismounted, and the Alpha AXP system rebooted. Providing file transfer support by means of the DECnet for OpenVMS AXP software early in the Alpha AXP project provided increased productivity for anyone testing on Alpha AXP prototype systems.

Porting Process

The process of porting the DECnet software from the VAX hardware platform to the Alpha AXP platform consisted of the following steps: code preparation, compilation, linking, code review, debug, and testing. We did not start the task of porting DECnet-VAX with a completely clear vision of the total process. As we progressed and learned more about the tools and porting process, we improved our porting techniques and, as a result, our productivity.

Our strategy was to begin by porting the drivers and privileged code. These components were the most complex; they were written completely in

MACRO-32 code and had the greatest potential for change. We started with NETDRIVER and NETACP, assigning one engineer to work on each component. As our porting group grew in number, we began to port, in parallel, the BLISS modules that comprise NCP, NML, NMLSHR, EVL, and MOM.

DECnet for OpenVMS AXP: A Case History

The following is an overview of the process we used to port the DECnet-VAX software to the Alpha AXP platform. Later sections contain details of coding practices that had to change.

Code Preparation. Our first task was to create procedures that we could use early in the porting process to compile single modules of a DECnet for OpenVMS AXP component. We also ported the component's build procedure to use the new Alpha AXP cross tools.

Next, we began to prepare the code for initial compilation. MACRO-32 code must have each entry point identified prior to the initial compile. Entry points are identified by a compiler directive such as .JSB_ENTRY and .CALL_ENTRY. Each directive accepts optional parameters that identify register usage. However, this information is not required at this point in the porting process. The Alpha AXP MACRO-32 compiler will provide register usage hints during the compilation, if so directed. As the team became familiar with the porting process, we were able to combine these steps and include the register usage information when declaring entry points. Also, as our experience increased, we were able to make changes to nonportable coding practices prior to the initial compile of a module.

Our experience proved, as we expected, that BLISS code is far easier to port than MACRO-32 code. For the DECnet-VAX components containing BLISS modules, we began the port by running the component's build procedure. BLISS routines do not require that entry points be identified. The compiler can process each module, identify errors, and provide warning and informational messages.

Compile Process. After we completed the initial code preparation and created customized build procedures, the real iterative process of porting began. At this point we compiled one or more modules, made additional modifications based on the compilation results, and recompiled until we were reasonably satisfied that all the errors were fixed.

The Alpha AXP cross compilers, the MACRO-32 compiler in particular, have the capability of providing a vast array of informational and warning messages. When compiling a module, we always requested all informational messages. The information assisted us in identifying the input and output registers as well as the registers that the compiler automatically preserved. Using this information, we verified the register usage in each routine and added the information to the entry-point directives. Other informational and warning messages directed us to coding techniques that required change. By working with one module at a time, we avoided making repetitive porting errors in multiple modules prior to our complete understanding of how to solve the more complex porting problems.

Some informational messages caution that certain coding techniques such as data alignment should be modified. We observed that attempting to make changes to align all data structure elements prior to completing preliminary debug and testing caused many debug problems. Therefore, we decided to establish a porting policy to change only as much code as was absolutely necessary prior to the initial debug and test of a DECnet for

OpenVMS AXP software component. Adhering to this policy required careful consideration, since some atomicity and granularity problems that are not resolved/addressed might cause code failures during debug.[3]

NETDRIVER and NETACP contained architecture-specific code, including memory management, driver tables, and structure definitions, which had to be made conditional for the OpenVMS AXP and OpenVMS VAX systems. A prefix file was added to each MACRO-32 module during the Alpha AXP compilation step. This file contained an Alpha AXP declaration that allowed us to include the directives required for conditional compilation. To compile the ported code on a VAX system, it was necessary to provide a VAX declaration and macros for the various entry-point directives that when expanded contained no instructions. These were placed in a common library file and conditionally compiled. The library file is included in each module. Figure 2 is an example of a library file that contains a VAX declaration and macros.

BLISS architecture-specific code was made conditional using the %if %bliss(bliss32v) or %if %bliss(bliss32e) constructs for OpenVMS VAX and OpenVMS AXP, respectively.

After porting all the modules within a component, the component's build procedure was run to ensure that each module had been ported without error. This was typically the first attempt to link the component. We also ran the OpenVMS VAX procedure to ensure that the code would continue to compile and link on the OpenVMS VAX operating system.

Linking. The process of linking was difficult at times. The DECnet for OpenVMS AXP software contains drivers, system images, and shareable images. Each component required changes to the link procedures. We made these procedures conditional for both the OpenVMS VAX and the OpenVMS AXP operating systems.

The process of linking the ported modules brought to light many unresolved references. In general, these references were to external routines that had changed for the OpenVMS AXP operating system. One of the most difficult aspects of the porting project was determining which changes to the OpenVMS operating system had an impact on our project. Determining these changes was difficult because DECnet for OpenVMS AXP is tightly integrated into the OpenVMS AXP operating system. The process of porting applications to the OpenVMS AXP environment should not be as difficult.

Code Review. When all the known porting problems found during the compile and link phases had been corrected, we began our code review process. The original VAX code, the ported code, and a difference listing were available to the porting team. One or more members of the team reviewed the changes made and pointed out any problems that were identified to the person responsible for the module being reviewed. We all had previously agreed that the reviews would be friendly and that egos would be left out of the process. We found that our successful code reviews were well worth the effort.

Initial reviews turned up differing philosophies regarding the porting process. We discussed these differences and reached a consensus. The reviews uncovered errors in the porting process, and correcting these problems reduced the amount of debugging required. The review process also allowed us to agree on and maintain coding standards.

Debugging. Our approach to debugging the DECnet for OpenVMS AXP software was to build the common ported code for a VAX system and to replace the OpenVMS VAX images with our ported version on one of our workstations. We began by loading the ported NETDRIVER and NETACP components. Since many of the required changes were common to both OpenVMS AXP and OpenVMS VAX systems, we were able to debug much of this code before we had access to Alpha AXP hardware. We found and fixed a number of problems using this technique.

When we were reasonably confident that the ported code was working on the OpenVMS VAX operating system, we began testing on Alpha AXP prototype hardware, which fortunately had just become available. We completed the driver load and ACP initialization testing. The initial test uncovered some problems that required special workarounds to allow debug to continue. These problems were corrected in later versions of the tools. Since the user interface had not yet been ported, test code was written to start DECnet for OpenVMS AXP and begin debugging the $QIO interface to the driver.

Eventually NCP, NML, and NMLSHR were ported, and more comprehensive debugging began. We used the OpenVMS AXP XDelta and Delta tools to debug the DECnet for OpenVMS AXP code on our Alpha AXP prototype hardware. System failures were debugged using the SDA cross tool on a VAX system. We learned how to trace call chains by studying the OpenVMS calling standard.[7] Understanding the format of linkage pairs, procedure descriptors, and register save areas made debugging much easier prior to the availability of these features in SDA. Debugging on an Alpha AXP system is more difficult than on a VAX system since most VAX instructions generate multiple Alpha AXP instructions whose positions are optimized by the compiler to take advantage of Alpha AXP architecture features. Thus, it is not always easy to follow the Alpha AXP code line by line because the generated Alpha AXP code from one language statement is interspersed with Alpha AXP code generated from another language statement.

Testing. After solving the obvious problems during the debug process, we began to test the DECnet for OpenVMS AXP code. Early versions of the OpenVMS AXP file system, record management services (RMS), and the file access listener (FAL) were made available to us. We in turn provided the DECnet for OpenVMS AXP code to the group porting OpenVMS RMS and FAL for their use in debugging. We were then able to run test scripts that used a variety of DCL commands to perform loops of remote copies, differences, and directory listings of remote files. DECnet network management scripts tested the network management interface. DTS and DTR were used to perform data transfer testing. Since the DECnet for OpenVMS AXP software was available early, it was used by other Alpha AXP porting groups on Alpha

AXP prototype hardware in various locations. As the code stabilized, a timesharing system was set up, which provided the opportunity for more testing.

Porting Issues

When we began porting the DECnet-VAX software to the Alpha AXP hardware platform, we found many coding conventions could not be used. Most of these coding practices are called out by the cross compilers, which significantly helped the porting effort.[3]

The following is a discussion of some problems we encountered while porting and how we solved them.

Entry Points. Approximately four months into the project, the porting team determined that using the .JSB_ENTRY directive in NETDRIVER was going to make porting difficult. The difficulty was due to the complexity of the code and the fact that some code paths contained more than a dozen layers of subroutine calls. We concluded that the code, which had existed for a long time, already saved and restored the correct registers. We decided that trying to communicate the correct list of input, output, pass-through, and preserve registers to the compiler could be an impossible task, especially given our schedule. We investigated the possibility of using the .JSB32_ENTRY directive. This directive allows the specification of registers that must be preserved but does not take any input, output, or scratch parameters. The OpenVMS AXP MACRO-32 cross compiler will not automatically preserve any registers when this directive is used. A great deal of care must be taken when using this entry-point directive.

Our decision to use .JSB32_ENTRY to declare entry points led to an interesting problem with asynchronously executing code that could interrupt other threads of execution. The DECnet-VAX code that we ported used PUSHR and POPR instructions to save and restore registers that were modified by DECnet-VAX code interrupting another thread of execution. When adding the .JSB32_ENTRY directives, we specified a register preserve parameter only on external entry points, assuming that the remainder of the original DECnet-VAX code was saving the proper registers. The preserve parameter ensures that all 64 bits of the registers specified are saved at routine entry and restored at routine exit. The PUSHR and POPR instructions preserve only the low-order 32 bits of the specified registers. However, if DECnet-VAX code in a routine without the .JSB32_ENTRY preserve parameter interrupts another thread of execution that makes use of the upper 32 bits of a register, these upper 32 bits would not be properly restored when control returned to the interrupted thread. The solution was to specify the register preserve parameter on the .JSB32_ENTRY directives used to declare the entry points of routines in DECnet for OpenVMS AXP that are capable of interrupting other threads of execution.

Whenever we changed the input or output parameters to an internal subroutine, we also changed the name of that subroutine. This effort helped identify all the internal calls made to subroutines whose interface had changed.

Coroutines. A feature of the VAX architecture used throughout the NETACP and NETDRIVER components is called a coroutine. Coroutines used in MACRO-32 allow a subroutine to call code fragments in other subroutines. This technique uses the jump-to-subroutine construct JSB @(SP)+ to jump between coroutines. The code example shown in Figure 3 demonstrates the use of the JSB construct.

The general flow of the example is for MAIN to call COROUTINE with R0 equal to 0 and R1 equal to 1. Usually, COROUTINE changes the value of R1 to 2 and calls back MAIN at address SAVE. If COROUTINE is entered with R1 not equal to 1, then R0 is set to 1 and the coroutine dialogue terminates. MAIN at address SAVE then tests R0 and exits. Under normal circumstances, MAIN at address SAVE continues, storing the returned value of R1 in DATA and calling back the coroutine at address FINAL. COROUTINE at address FINAL then changes the value of R1 to 3, sets the return status in R0 to 1, and returns to MAIN at address TERMINATE. TERMINATE then exits MAIN via the RSB instruction.

All entry points in MACRO-32 code on an OpenVMS AXP operating system must have an entry directive. Thus, it is not possible to use the JSB construct to jump to any random line of code, as the previous example demonstrates. To do so, the code shown in Figure 3 would have to be split into subroutines, each with a .JSB_ENTRY or .JSB32_ENTRY entry directive. Also, we had to change the implementation of coroutines. Rather than use the stack to pass return addresses, we passed each return address in a register.

Since some coroutines ported were more complex than the example shown in Figure 3, we developed a technique to port VAX coroutines to the Alpha AXP architecture. When a coroutine is split into multiple routines, some code, such as that testing returned values, may change relative location. In our example, the error processing at SAVE is no longer necessary. Instead, COROUTINE returns to MAIN if it detects an error, and MAIN simply returns to its caller with the status in R0. The VAX code example in Figure 3 was converted to Alpha AXP code using our technique. The resulting code is shown in Figure 4.

The use of coroutines on Alpha AXP systems should be discouraged because of the overhead associated with storing the return address in registers and the additional consumption of stack space. Rather than a simple return address on the stack, there will be a register save area on the stack for each subroutine that makes up the coroutine. Recursive coroutines can consume large quantities of stack space. We have since converted coroutines used in main code paths to straight in-line subroutine calls.

Stack Usage. MACRO-32 code uses a number of common coding techniques that require knowledge of the state of the stack and that must be changed for the OpenVMS AXP operating system. One such technique, referred to as an up-level stack reference, occurs whenever a subroutine attempts to access information (address or data) stored on the stack by its caller. Parameter passing sometimes uses this technique. If a routine pushes arguments onto the stack prior to jumping to a subroutine, the called subroutine does up-level stack references to retrieve the arguments. Other techniques include using the stack as a common data area or attempting to manipulate the caller's return address in order to alter the program flow.

All these techniques require re-coding. When we encountered code that passed parameters on the stack, we modified the code to pass parameters in registers. If a structure was being passed, separate memory was allocated and the address of the structure passed in a register. In one case, NETACP used coroutines to perform specific functions to update a common data area allocated on the stack. This code was redesigned to eliminate the coroutines and up-level stack references. Another alternative would have been to pass the address of the data area on the stack to the called routine.

Altering the program flow when error conditions were encountered was also a common technique used in the DECnet-VAX MACRO-32 code. Subroutines removed the return address from the stack and returned to the caller's caller. We modified the code to remove the up-level stack reference (the caller's return address) and return a flag in a register to signal the caller that a change in the program flow was desired.

Condition Codes. The Alpha AXP architecture does not support global condition codes in the processor status word. Some routines set condition codes and returned to the caller, which proceeded to perform a conditional branch on the results of the called routine. All occurrences of this technique were changed; routines now pass the result of any conditional test to the caller in a register.

Granularity and Atomicity Issues.[8] The NETACP and NETDRIVER components access shared data structures. Since NETDRIVER can interrupt NETACP, the DECnet-VAX code relies on the atomicity of VAX instructions to provide synchronized access to shared fields in the data structures. The DECnet-VAX code also relies on byte (8-bit) and word (16-bit) granularity for memory writes. Since the granularity of Alpha AXP memory writes is either longword (32-bit) or quadword (64-bit), DECnet-VAX code that required atomic access to word fields had to be modified to protect against writes to neighboring byte and word fields sharing the same longword or quadword. In DECnet for OpenVMS AXP, word data structure fields shared by NETACP and NETDRIVER that required atomic access were moved to their own aligned quadwords to prevent interference from simultaneous writes to other byte and word fields sharing the same quadword. After the word fields were placed in their own aligned quadwords, the code generated by the MACRO-32 cross compiler for the ADAWI instruction was sufficient to provide atomic access to the word fields. We could also have used compiler directives to specify that VAX granularity and atomicity rules be preserved.

BLISS-32 Code. The BLISS-32 code in the DECnet-VAX software was relatively simple to port. We made minor changes to add the VOLATILE parameter to data items that should not be cached in registers, to conditionally compile the exception handlers for VAX or Alpha AXP, and to remove unsupported built-ins. Other modifications were more extensive, such as the changes to accommodate the new LIB$TABLE_PARSE.

LIB$TPARSE Changes. LIB$TPARSE and LIB$TABLE_PARSE are the interface routines to a general-purpose, table-driven parser for the OpenVMS VAX and OpenVMS AXP operating systems, respectively. The call to these routines was made conditional for the VAX and Alpha AXP architectures. Other changes were required because LIB$TPARSE and LIB$TABLE_PARSE differ in the way argument lists are passed. The method used by LIB$TPARSE to pass arguments is incompatible with the OpenVMS AXP calling standard. The LIB$TPARSE action routines required modification as a result of the required change to LIB$TABLE_PARSE for the OpenVMS AXP operating system. The LIB$TPARSE action routines received all or a subset of the argument block as parameters. LIB$TABLE_PARSE passes the address of the argument block to the action routines. The solution we used was to make the routine declaration conditional so that on the OpenVMS VAX operating system the action routines continued to receive the argument block parameters, and on the OpenVMS AXP operating system the action routines received the address of the argument block. Next, for the OpenVMS AXP operating system, the parameter names used by the common code were bound to the argument block. These changes are shown in Figure 5.

As a result of this relatively simple though repetitive change, no other changes had to be made in the action routines. If at some future time the OpenVMS VAX operating system uses LIB$TABLE_PARSE, there will be no need for conditionals.

4 Conclusion

This porting effort not only provided a solid base of knowledge with which to begin the port of the DECnet/OSI for OpenVMS VAX software and the associated products, but also gave us an appreciation of common code and the avoidance of architecture-specific code.

More and more software is being ported to new hardware platforms. The porting process is often carried out by individuals who did not develop the original software and who may not even be familiar with it. Our experience porting the DECnet-VAX code leads us to believe that new software development should take into account the possibility that the code will be ported to new hardware platforms at some future date. As we continue to port the DECnet/OSI for OpenVMS VAX software, it is becoming increasingly obvious that certain coding practices are difficult to port. As a general suggestion, if the code has knowledge of the architecture but can be written using system routines, system services, or run-time library functions, write the code in that manner. These system routines will be ported with the operating system, and in a majority of the cases, the application code will not require modification.

Also, if architecture-specific functions are required, provide only a minimum amount of code to perform these required functions and segregate the code. Document how the code works, why it had to be done that way, what the alternatives were, and why they were not taken. In addition to helping maintain the code, this information may provide valuable assistance to a person porting the code in the future.

If a routine is written in assembly language for the sole purpose of performance improvement, consider rewriting it in a high-level language. It is possible that the assembly language coding conventions that may have been optimal for one hardware platform will be slower on a different hardware platform. Using high-level language compilers, which generate optimized hardware-specific code, will eliminate additional porting effort and may very likely be the best performance solution.

As we discovered during the process of porting the DECnet-VAX software, MACRO-32 code is significantly more difficult to port than code written in higher-level languages. However, certain architecture-specific functions may have to be written in assembly language. We recommend that these functions be isolated. In addition, we recommend that any other code written in MACRO-32 be rewritten, over time, in a higher-level language.

We determined that the fastest approach to porting was to make the minimum number of changes required to get the DECnet for OpenVMS AXP software running. The porting process was accomplished in phases. The first phase included the initial port and addressed any errors that occurred until we successfully completed linking the image. This phase also included the initial debug, which was first performed on VAX systems because of our common code approach and, subsequently, done on Alpha AXP prototype hardware. When the product was stable, we proceeded to the second phase in which we began to methodically align data structures and fix granularity and atomicity problems. Small changes could then be made and tested, and any new problems were generally easy to identify.

Our team approach to the project worked extremely well. Each team member was initially responsible for porting specific portions of the code. As the project progressed, individuals worked on any part of the product that needed attention. This flexibility was also used when we began to debug the ported code and again when we began to respond to problem reports. Priorities were used to assign resources in order to solve problems as quickly as possible. Throughout the project, team members worked together to share knowledge and to solve problems efficiently. This effective teamwork allowed us to deliver the DECnet for OpenVMS AXP product ahead of schedule.

5 Acknowledgments

The authors would like to thank the other members of the software development team, Ken Roberts, Cathy Wright, our manager John Heron, and the group engineering manager Morea Martocchio, whose hard work made this project a success. In addition, we would like to thank all the individuals of the Alpha AXP project who helped us along the way. In particular, we would like to recognize certain individuals for their important contributions to the success of this project: Paul Weiss, our porting consultant; Lenny Scubowitz, David Gagne, and Ben Thomas of the I/O team; Karen Noel and Mike Harvey of the executive group; and Steve Dipirro of the XDelta team.

The DECnet for OpenVMS AXP project was only part of the Alpha AXP team effort. We feel fortunate to have experienced the synergy that this team created.

6 References

1. A. Lauck, D. Oran, and R. Perlman, "Digital Network Architecture Overview," Digital Technical Journal, vol. 1, no. 3 (September 1986): 10-24.

2. P. Beck and J. Krycka, "The DECnet-VAX Product - An Integrated Approach to Networking," Digital Technical Journal, vol. 1, no. 3 (September 1986): 88-99.

3. Migrating to an OpenVMS Alpha System: Porting VAX MACRO Code (Maynard: Digital Equipment Corporation, Order No. AA-PQYEA-TE, 1992).

4. OpenVMS Linker Manual (Maynard: Digital Equipment Corporation, Order No. AA-PQXYA-TK, 1992).

5. OpenVMS Alpha System Dump Analyzer Utility Manual (Maynard: Digital Equipment Corporation, Order No. AA-PQYCA-TE, 1992).

6. OpenVMS Delta/XDelta Utility Manual (Maynard: Digital Equipment Corporation, Order No. AA-PQYPA-TK, 1992).

7. OpenVMS Calling Standard (Maynard: Digital Equipment Corporation, Order No. AA-PQY2A-TK, 1992).

8. N. Kronenberg et al., "Porting OpenVMS from VAX to Alpha AXP," Digital Technical Journal, vol. 4, no. 4 (1992, this issue): 111-120.

7 General References

DECnet for OpenVMS Network Management Utilities (Maynard: Digital Equipment Corporation, Order No. AA-PQYAA-TK, 1992).

DECnet for OpenVMS Guide to Networking (Maynard: Digital Equipment Corporation, Order No. AA-PQY8A-TK, 1992).

DECnet for OpenVMS Networking Manual (Maynard: Digital Equipment Corporation, Order No. AA-PQY9A-TK, 1992).

Migrating to an OpenVMS Alpha System: Planning for Migration (Maynard: Digital Equipment Corporation, Order No. AA-PQY7A-TE, 1992).

8 Trademarks

9 Biographies

James V. Colombo Project/technical leader James Colombo is currently responsible for the next release of DECnet/OSI for OpenVMS for the VAX and Alpha AXP computing environments. Prior to this, he led the port of DECnet-VAX Phase IV to the OpenVMS AXP operating system; the team received an Alpha Achievement Award for early completion of the project. Jim also led the DECnet for OS/2 V1.0 and various PATHWORKS product efforts. Before coming to Digital in 1983, Jim worked at Prime Computer, Inc. and Computer Devices, Inc. He holds a B.S.C.S. from Boston University and is a member of IEEE.

Pamela J. Rickard Principal software engineer Pam Rickard is a member of the team porting DECnet/OSI for OpenVMS to the Alpha AXP platform. As the initial member of the DECnet for OpenVMS AXP porting team, Pam took responsibility for creating an effective team, ported NETDRIVER and other MACRO-32 code, and debugged major portions of the ported product. Since joining Digital in 1978, she has contributed to PATHWORKS for OS/2 and led the console, microcode, and system test activities of the VAX-11/785 project. Pam received a B.S. (1970) in mathematics and computer science from the University of Denver.

Paul Benoit Paul Benoit is a principal software engineer in the Networks and Communications Group. He is the project/technical leader for the DECnet for OpenVMS AXP project; the team received an Alpha Achievement Award for early completion of project commitments. Previous to this, Paul led the DECnet-VAX Phase IV effort. He holds an M.S.S.E. (1991) from Boston University and a B.S.C.S. (1986) from the University of Lowell. Paul is a member of ACM and IEEE Computer Society.