Porting OpenVMS from VAX to Alpha AXP

1 Abstract

The OpenVMS operating system, developed by Digital for the VAX family of computers, was recently moved from the VAX to the Alpha AXP architecture. The Alpha AXP architecture is a new RISC architecture introduced by Digital in 1992. This paper describes solutions to several problems in porting the operating system, in addition to performance benefits measured on one of the systems that implements this new architecture.

The VAX architecture is an example of complex instruction set computing (CISC), whereas the Alpha AXP architecture is based on reduced instruction set computing (RISC). The two architectures are very different.[1] CISC architectures have performance disadvantages as compared to RISC architectures.[2] Digital ported the OpenVMS system to the Alpha AXP architecture mainly to deliver the performance advantages of RISC to OpenVMS applications. This paper focuses on how Digital's OpenVMS AXP operating system group dealt with the large volume of VAX assembly language and with system kernel modifications that had VAX architecture dependencies.

The OpenVMS AXP group had two important requirements in addition to the primary goal of increasing performance: first, to make it easy to move existing users and applications from OpenVMS VAX to OpenVMS AXP systems; second, to deliver a high-quality first version of the product as early as possible. These requirements led us to adopt a fairly straightforward porting strategy with minimal redesigns or rewrites. We view the first version of the OpenVMS AXP product as a beginning, with other evolutionary steps to follow.

The Alpha AXP architecture was designed for high performance but also with software migration from the VAX to the Alpha AXP architecture in mind. Included in the Alpha AXP architecture are some VAX features that ease the migration without compromising hardware performance. VAX features in the Alpha AXP architecture that are important to OpenVMS system software are: four protection modes, per-page protection, and 32 interrupt priority levels. The Alpha AXP architecture also defines a privileged architecture library (PAL) environment, which runs with interrupts disabled and in the most privileged of the four modes (kernel). PALcode is a set of Alpha AXP instructions that executes in the PAL environment, implementing such basic system software functions as translation buffer (TB) miss service. On OpenVMS AXP systems, PALcode also implements some OpenVMS VAX features such as software interrupts and asynchronous traps (ASTs). The combination of hardware architecture assists and OpenVMS AXP PALcode made it easier to port the operating system to the Alpha AXP architecture.

The VAX architecture is 32-bit; it has 32 bits of virtual address space, 16 32-bit registers, and a comprehensive set of byte, word (16-bit), and longword (32-bit) instructions. The Alpha AXP architecture is 64-bit, with 64 bits of virtual address space, 64-bit registers (32 integer and 32 floating-point), and instructions that load, store, and operate on 64-bit quantities. There are also longword load, store, and operate instructions, and a canonical sign-extended form for a longword in a 64-bit register.

The OpenVMS AXP system has anticipated evolution from 32-bit address space size to 64-bit address space by changing to a page table format that supports large address space. However, the OpenVMS software assumes that an address is the same size as a longword integer. The same assumption can exist in applications. Therefore, the first version of the OpenVMS AXP system supports 32-bit address space only.

Most of the OpenVMS kernel is in VAX assembly language (VAX MACRO-32). Instead of rewriting the VAX MACRO-32 code in another language, we developed a compiler. In addition, we required inspection and manual modification of the VAX MACRO-32 code to deal with certain VAX architectural dependencies. Parts of the kernel that depended heavily on the VAX architecture were rewritten, but this was a small percentage of the total volume of VAX MACRO-32 source code.

2 Compiling VAX MACRO-32 Code for the Alpha AXP Architecture

Simply stated, the VAX MACRO-32 compiler treats VAX MACRO-32 as a source language to be compiled and creates native OpenVMS AXP object files just as a FORTRAN compiler might. This task is far more complex than a simple instruction-by-instruction translation because of fundamental differences in the architectures, and because source code frequently contains assumptions about the VAX architecture and the OpenVMS Calling Standard on VAX systems.[3,4] The compiler must either transparently convert these VAX dependencies to their OpenVMS AXP counterparts or inform the user that the source code has to be changed.

Source Code Annotation

We extended the VAX MACRO-32 source language to include entry-point declarations and other directives for the compiler's use, which provide more information about the intended behavior of the program. Entry-point declarations were introduced to allow declaration of: (1) the register semantics for a routine when the defaults are not appropriate and (2) the specialized semantics of frameless subroutines and exception routines to be declared.

The differing register size between the VAX and the Alpha AXP architectures influenced the design of the compiler. On the VAX, MACRO-32 operates on

32-bit registers, and in general, the compiled code maintains 32-bit sign-extended values in Alpha AXP 64-bit registers. However, this code is now part of a system that uses true 64-bit values. As a result, we designed the compiler to generate 64-bit register saves of any registers modified

in a routine, as part of the "routine prologue code." Automatic register preservation has proven to be the safest default but must be overridden for routines that return values in registers, using appropriate entry-point declarations.

Other directives allow the user to provide additional information about register state and code flow to improve generated code. Another class of directives instructs the compiler to preserve the VAX behavior with respect to granularity of memory writes or atomicity of memory updates. The Alpha AXP architecture makes atomic updates and guaranteed write granularity sufficiently costly to performance that they should be enabled only when required. These concepts are discussed in the section Major Architectural Differences in the OpenVMS Kernel.

Identifying VAX Architecture and Calling Standard Dependencies

As mentioned earlier, the compiler must either transparently support VAX architecture-dependent constructs or inform the user that a source change is necessary. A good example of the latter case is reliance on VAX JSB/RSB (jump to subroutine and return) instruction return address semantics. On VAX systems, a JSB instruction leaves the return address on top of the stack, which is used by the RSB instruction to return.[3] System subroutines often take advantage of this semantic in order to change the return address. This level of stack control is not available in a compiled language. In porting the OpenVMS system to the Alpha AXP architecture, we developed alternative coding practices for this and many other nontransportable idioms.

The compiler must also account for the differences in the OpenVMS Calling Standard on the VAX and Alpha AXP architectures. Although transparent to high-level language programmers, these differences are very significant in assembly language programming.[4] To operate correctly in a mixed language environment, the code generated by the VAX MACRO-32 compiler must conform to the OpenVMS Calling Standard on the Alpha AXP architecture.

On VAX systems, a routine refers to its arguments by means of an argument pointer (AP) register, which points to an argument list that was built in memory by the routine's caller. On Alpha AXP systems, up to six routine arguments are passed to the called routine in registers; any additional arguments are passed in stack locations. Normally, the VAX MACRO-32 compiler transparently converts AP-based references to their correct Alpha AXP locations and converts the code that builds the list to initialize the arguments correctly. In some cases, the compiler cannot convert all references to their new locations, so an emulated VAX argument list must be constructed from the arguments received in the registers. This so-called "homing" of the argument list is required if the routine contains indexed references into the argument list or stores or passes the address of an

argument list element to another routine.

The compiler identifies these coding practices during its process of flow analysis, which is similar to the analysis done by a standard high-level language optimizing compiler. The compiler builds a flow graph for each routine and tracks stack depth, register use, condition code use, and loop depth through all paths in the routine flow. This same information is required for generating correct and efficient code.

Access to Alpha AXP Instructions and Registers

In addition to providing migration of existing VAX code, the VAX MACRO-32 compiler includes support for a subset of Alpha AXP instructions and PALcode calls and access to the 16 integer registers beyond those that map to the VAX register set. The instructions supported either have no direct counterpart in the VAX architecture or are required for efficient operation on a full 64-bit register value. These "built-ins" were required because the OpenVMS AXP system uses full 64-bit values for some operations, such as manipulation of 64-bit page table entries (PTEs).

Optimization

The compiler includes certain optimizations that are particularly important for the Alpha AXP architecture. For example, on a VAX system, a reference to an external symbol would not be considered expensive. On an Alpha AXP system, however, such a reference requires a load from the linkage section to obtain the address of the symbol prior to loading the symbol's value. (The linkage section is a data region used for resolving external references.[4]) Multiple loads of this address from the linkage section may be reduced to a single load, or the load may be moved out of a loop to reduce memory references.

Other optimizations include the elimination of memory reads on multiple safe references, register state tracking for optimal register-based memory references, redundant register save/restore removal, and many local code generation optimizations for particular operand types. Peephole optimization of local code sequences and low-level instruction scheduling are performed by the back end of the compiler.

In some instances, the programmer has knowledge of register state or other code behavior that cannot be inferred from the source code alone. Compiler directives are provided for specifying register alignment state, structure base address alignment, and likely flow paths at branch points.

Certain types of optimization typically performed by a high-level language compiler cannot be performed by the VAX MACRO-32 compiler, because assumptions made by the MACRO-32 programmer cannot be broken. For example, the order of memory reads may not be changed.

3 Major Architectural Differences in the OpenVMS Kernel

This section concentrates on architectural changes that affect synchronization, memory management, and I/O. These are not the only architectural differences that cause significant changes in the kernel. However, they are the major differences and are representative of the effort involved in porting the OpenVMS system to the Alpha AXP architecture.

For the most part, it was possible to isolate architecture-dependent changes to a few major subsystems. However, differences in the memory reference architecture had a pervasive impact on users of shared data and many common synchronization techniques. Other differences such as those related to memory management, context switching, or access to I/O devices were confined mostly to the relevant subsystems.

Effects of Architectural Differences in Memory Subsystems

The following differences between the VAX and Alpha AXP memory reference architectures affected synchronization:[1,3]

- Load/store architecture rather than atomic modify instructions
- Longword and quadword writes with no byte write instructions
- Read/write ordering not guaranteed

Load/store memory reference instructions are characteristic of most RISC designs. However, the other differences are less typical. The reasons for all three differences were hardware simplification and opportunities for increased hardware performance.[1] These differences result in significant changes in system software and in many opportunities for subtle errors, which can be detected only under heavy load. Adapting to these architectural changes without greatly impacting performance was one of the major challenges that faced the group in porting the OpenVMS system to the Alpha AXP architecture.

A load/store architecture such as Alpha AXP cannot provide the atomic read-modify-write instructions present in the VAX architecture. Thus, instruction sequences are necessary for many operations performed by a single, atomic VAX instruction, such as incrementing a memory location. The consequence is a need for increased awareness of synchronization. Instead of depending on hardware to prevent interference between multiple threads of execution on a single processor, explicit software synchronization may be required. Without this synchronization, the interleaving of independent load-modify-store sequences to a single memory location may result in overwritten stores and other incorrect results.

The lack of byte writes imposes additional synchronization burdens on software. Unlike VAX and most RISC systems, an Alpha AXP system has instructions to write only longwords and 64-bit quadwords, not bytes or words. Thus to write bytes, the software must include a sequence of instructions that reads the encompassing longword, merges in the byte, and

writes the longword to memory. As a consequence, software must be concerned not only with shared access to the same memory cell by multiple threads, but also with access to independent but adjacent variables. Synchronization awareness is now extended from shared data to data items that are merely near each other.

The OpenVMS AXP operating system group avoided the above-mentioned problems introduced by the architectural changes in one of three ways:

- Explicit software synchronization was added between threads.
- Data items were relocated to aligned longwords or quadwords.
- Alpha AXP load locked and store conditional instructions were used.

The obvious solution of adding explicit synchronization in the form of a software lock is not always appropriate for several reasons. First, the problem may be independent data items that happen to share a longword. Synchronizing this sort of access in unrelated code paths is prone to error. Explicit synchronization may also have an unacceptable performance impact. Finally, deadlocks are a possibility if one thread interrupts another.

Ensuring that data items are in aligned longwords or quadwords both improves performance and eliminates interactions between unrelated data. This technique is used wherever possible but cannot be used in two major cases. One case occurs when the replication factor is too large. Expanding an array of thousands of bytes to longwords may simply not be acceptable. The other major problem case is data structures that cannot be changed because of external constraints. For example, data may represent a protocol message or a structure primarily resident on disk. Separate internal and external forms of such data structures could exist, but the performance cost of continuous conversions may not be acceptable.

Often the easiest and highest-performance way to solve synchronization problems is to use sequences of load locked and store conditional instructions. The load locked instruction loads an aligned longword or quadword while setting a hardware flag that indicates the physical address that was loaded. The hardware flag is cleared if any other thread, processor, or I/O device writes to the locked memory location. The store conditional instruction stores an aligned longword or quadword if and only if the hardware lock flag is still set. Otherwise, the store returns an error indication without modifying the storage location. These instructions allow the construction of atomic read-modify-write sequences to update any datum that is contained within a single aligned quadword. These sequences of instructions are significantly slower than normal loads and stores due to the necessity of waiting for the write to reach a point in the memory

hierarchy where consistency can be guaranteed. In addition, their use may inhibit many compiler optimizations because of restrictions on the instructions between the load and store. Although faster than most other synchronization methods, this mechanism should be used sparingly.

The lack of guaranteed read/write ordering between multiple processors is another complication for the programmer trying to achieve proper synchronization. Although not visible on a single processor, this lack of ordering means that one processor will not necessarily observe memory operations in the order in which they were issued by another processor. Thus, many obvious synchronization protocols will not work when writes to the synchronization variable and to the data being protected are observed to occur out of order. A memory barrier instruction is provided in the architecture to ensure ordering. However, the negative impact of this instruction on system performance requires that it be used only when necessary.

As described in the previous section, we used various mechanisms to solve the synchronization problems. Having multiple solutions allowed us to choose the one with the least performance impact for each case. In some cases, explicit synchronization was already in place due to multiprocessor synchronization requirements. In other cases, we expanded data structures at a cost of modest amounts of memory to avoid expensive synchronization when referencing data.

Privileged Architecture Changes

Unlike the pervasive architectural changes described in the previous section, the privileged architecture differences had a more limited impact. The primary remaining areas of change are the new page table formats and the details of process context switching. This section describes memory management as a representative example. However, many limited changes still amount to modifying virtually every source module in the OpenVMS kernel, even if only to add compiler directives.

Memory Management. Not surprisingly, the memory management subsystem required the most change when moving from the VAX to the Alpha AXP architecture. Aside from the obvious 64-bit addressing capability, two significant differences exist between the page table structures on the VAX and the Alpha AXP architectures. First, Alpha AXP does not have an architectural division between shared and process private address space. Second, the Alpha AXP three-level page table structure shown in Figure 1 allows the sharing of arbitrary subtrees of the page table structure and the efficient creation of large, sparse address spaces. In addition, future Alpha AXP processors may have larger page sizes.

To meet our schedule goals, we decided initially to emulate the VAX architecture's 32-bit address space as closely as possible. This decision required creating a 2-gigabyte (GB) process private address region (i.e., VAX P0 and P1) and a 2GB shared address region (i.e., VAX S0 and S1) for each process. This is easily accomplished by giving each process a private level 1 page table (L1PT) that contains two entries for level 2 page tables

(L2PTs). One of these L2PTs is shared and implements the shared system region, whereas the other is private and implements the process private address regions. Although the smallest allowed page size of 8 kilobytes (KB) results in an 8GB region for each level 2 page table, we use only 2GB

of each region to keep within our 4GB 32-bit limit. As shown in Figure 1, the L2PTs are chosen to place the base address of the shared system region at FFFFFFFF80000000 (hexadecimal), the same as the sign-extended address of the top half of the VAX architecture's 32-bit address space.

Although changes were extensive in the memory management subsystem, many were not conceptually difficult. Once we dealt with the new page table structure, most changes were merely for alternative page sizes, new page table entry formats, and changes to associated data structures. We did decide to keep the OpenVMS VAX concept of mapping process page tables as a single array in shared system space for our initial implementation. Although not viable in the long term due to the potentially huge size of sparse process page tables, this decision minimized changes to code that references process page tables. Having process page tables visible in shared system space turned out to be a significant complication in paging and in address space creation, but the schedule benefits derived from avoiding changes to other subsystems were considered worthwhile. We expect to change to a more general mechanism of self-mapping process page tables in process space for a subsequent OpenVMS AXP release.

Retaining the VAX appearance of process page tables allowed us to meet our goals of minimum change outside of the memory management subsystem. Unprivileged code is unaffected by the memory management changes unless it is sensitive to the new page size. Even privileged code is generally unaffected unless it has knowledge of the length or format of PTEs.

Optimized Translation Buffer Use. Thus far, we may have given the impression that architectural changes always create problems for software. This was not universally true; some changes offered us opportunities for significant gains. One such change was an Alpha AXP translation buffer feature called granularity hints. TBs are key to performance on any virtual memory system. Without them, it would be necessary to reference main memory page tables to translate every virtual address to a physical address. However, there never seems to be enough TB entries. The Alpha AXP architecture allows a single TB entry to optionally map a virtually and physically contiguous block of properly aligned pages, all with identical protection attributes. These pages are marked for the hardware by a flag in the PTE.

Given granularity hints, near-zero TB miss rates for the kernel became attainable. To this end, we enhanced the kernel loading mechanisms to create and use granularity hint regions.

The OpenVMS AXP kernel is made up of many separate images, each of which contains several regions of memory with varying protections. For example, there is read-only code, read-only data, and read-write data. Normally, a kernel image is loaded virtually contiguously and relocated so that it can

execute at any address. To take advantage of granularity hints, kernel code and data are loaded in pieces and relocated to execute from discontiguous regions of memory. Only a very few TB entries are actually used to map the entire nonpaged kernel, and the goal of near-zero TB misses was reached.

The benefits of granularity hints became immediately obvious, and the mechanism has since been expanded. Now, the OpenVMS AXP system also uses the code region for user images and libraries. This feature extends the benefits not only to images supplied by the OpenVMS system, but to customer applications and layered products as well. Of course, usage of this feature is only reasonable for images and libraries used so frequently that the permanent allocation of physical memory is warranted. However, most applications are likely to have such images, and the performance advantage can be impressive.

I/O

Unlike the architectural changes, the new I/O architecture structures an area that previously was rather uncontrolled. The project goal was to allow more flexibility in defining hardware I/O systems while presenting software with a consistent interface. These seem like contradictory aims, but both must be met to build a range of competitive, high-performance hardware that can be supported with a reasonable software effort.

The Alpha AXP architecture presents a number of differences and challenges that impacted the OpenVMS AXP I/O system. These changes spanned areas from longword granularity to device control and status register (CSR) access to how adapters may be built.

CSR Access. A fundamental element of I/O is the access of CSRs. On VAX systems, CSR access is accomplished as simply another memory reference that is subject to a few restrictions. Alpha AXP systems present a variety of CSR access models.

Early in the project, the concept of I/O mailboxes was developed as an alternative CSR access model. The I/O mailbox is basically an aligned piece of memory that describes the intended CSR access. Instead of referencing CSRs by means of instructions, an I/O mailbox is constructed and used as a command packet to an I/O processor. The mailbox solves three problems: the mailbox allows access to an I/O address space larger than the address space of the system; byte and word references may be specified; and the system bus is simplified by not having to accommodate CSR references that may stall the bus. As systems get faster, these bus stalls are increasingly larger impediments to performance.

Mailboxes are the I/O access mechanism on some, but not all, systems. To preserve investment in driver software, the OpenVMS AXP operating system implemented a number of routines that allow all drivers to be coded as if CSRs were accessed by a mailbox. Systems that do not support mailbox I/O have routines that emulate the access. These routines provide insulation from hardware implementation details at the cost of a slight performance impact. Drivers may be written once and used on a number of differing

systems.

Read/Write Ordering. An I/O device is simply another processor, and the sharing of data is a multiprocessing issue. Since the Alpha AXP architecture does not provide strict read/write ordering, a number of rules must be followed to prevent incorrect behavior. One of the easiest changes is to use the memory barrier instructions to force ordering. Driver code was modified to insert memory barriers where appropriate.

The devices and adapters must also follow these rules and enforce proper ordering in their interactions with the host. An example is the requirement that an interrupt also act like a memory barrier in providing ordering. In addition, the device must ensure proper ordering for access to shared data and direct memory access.

Kernel Processes. Another important way in which the Alpha AXP architecture differs from the VAX architecture is the lack of an interrupt stack. On VAX systems, the interrupt stack is a separate stack for system context. With the new Alpha AXP design, any system code must use the kernel stack of the current process. Therefore, a process kernel stack must be large enough for the process and for any nested system activity. This burden is unreasonable. A second problem is that the VAX I/O subsystem depends on absolute stack control to implement threads. As a result, most of the I/O code is in MACRO-32, which is a compiled language on the OpenVMS AXP system that does not provide absolute stack control.

These facts resulted in the creation of a kernel threading package for system code at elevated interrupt priority levels. This package, called kernel processes, provides a set of routines that support a private stack for any given thread of execution. The routines include support for starting, terminating, suspending, and resuming a thread of execution.

The private stack is managed and preserved across the suspension with no special measures on the part of the execution thread. Removing requirements for absolute stack control will facilitate the introduction of high-level languages into the I/O system.

4 Performance

As stated earlier, the main purpose of the project was to deliver the performance advantages of RISC to OpenVMS applications. We adopted several methods, including simulation, trace analysis, and a variety of measurements, to track and improve operating system and application performance. This section presents data on the performance of OpenVMS services and on the SPEC Release 1 benchmark suite.[5] Note that all Alpha AXP results are preliminary.

Performance of OpenVMS Services

To evaluate the performance of the OpenVMS system, we used a set of tests that measure the CPU processing time of a range of OpenVMS services. These tests are neither exhaustive nor representative of any particular workload. We use relative CPU speed (i.e., VAX CPU time divided by Alpha AXP CPU time) as a metric to truly compare CPU performance. For I/O-related services, a RAM disk was used to eliminate I/O latencies.

The tests were run on a VAX system and an Alpha AXP system that are the same except for the CPU. Table 1 shows the configuration details of the two systems. Figure 2 shows the distribution of the relative CPU speed for the OpenVMS services measured. Most tests ran between 0.9 and 1.7 times faster on the Alpha AXP system than on the VAX system. Table 2 contains the results for a representative subset of the measured OpenVMS services.

Application Performance

Applications vary in their use of operating system services. Most applications spend the majority of their time performing application-specific work and a small fraction of their time using operating system services. For these applications, performance depends mainly on the performance of hardware, compilers, and run-time libraries. We used the SPEC Release 1 benchmarks as representative of such applications. Table 3 shows the details of the VAX and Alpha AXP systems on which the SPEC Release 1 suite was run, and Table 4 contains the results. The SPECmark89 comparison shows that the OpenVMS AXP system outperforms the OpenVMS VAX system by a factor of 3.59.

The performance of OpenVMS services and the SPECmark results are consistent with other studies of how operating system primitives and SPECmark results scale between CISC and RISC.[6] Overall, the results are very encouraging for a first-version product in which redesigns were purposely limited to meet an aggressive schedule.

5 Conclusions

Some OpenVMS VAX features such as symmetric multiprocessing and VMScluster support were deferred from the first version of the OpenVMS AXP system. Beyond this, we anticipate taking significant steps to better exploit the hardware architecture, including evolving to a true 64-bit operating system in a staged fashion. Also, detailed analysis of performance results shows the need to alter internal designs to better match RISC architecture. Finally, a gradual replacement of VAX MACRO-32 source with a high-level language is essential to support a 64-bit virtual address space and is an important element for increasing performance.

The OpenVMS AXP system clearly demonstrates the viability of making dramatic changes in the fundamental assumptions of a mature operating system while preserving the investment in software layered on the system. The future challenge is to continue operating system evolution in order to provide more capabilities to applications while maintaining that essential level of compatibility.

6 Acknowledgments

The work described in this paper was done by members of the OpenVMS AXP operating system group. This work would have been impossible without the help of many software and hardware engineering groups at Digital. Thanks to Bradley Waters, who measured OpenVMS performance, and to John Shakshober and Sandeep Deshmukh, who obtained the SPEC Release 1 benchmark results. We also thank Barbara A. Heath and Kathleen D. Morse for their comments, which helped in preparing this paper.

7 References

1. R. Sites, "Alpha AXP Architecture," Digital Technical Journal, vol. 4, no. 4 (1992, this issue): 19-34.

2. D. Bhandarkar and D. Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-IV) (New York, NY: The Association for Computing Machinery, 1991): 310-319.

3. T. Leonard, ed., VAX Architecture Reference Manual (Bedford, MA: Digital Press, 1987).

4. OpenVMS Calling Standard (Maynard, MA: Digital Equipment Corporation, October 1992).

5. Spec Newsletter, vol. 4, no. 1 (March 1992).

6. T. Anderson, H. Levy, B. Bershad, and E. Lazowska, "The Interaction of Architecture and Operating System Design," Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-IV) (New York, NY: The Association for Computing Machinery, 1991): 108-120.

8 General References

R. Goldenberg and S. Saravanan, VMS for Alpha Platforms Internals and Data Structures, Preliminary edition of vols. 1 and 2 (Maynard, MA: Digital Press, 1993, forthcoming).

J. Hennessy and D. Patterson, Computer Architecture, A Quantitative Approach (San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990).

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

9 Trademarks

10 Biographies

Nancy P. Kronenberg Nancy Kronenberg joined Digital in 1978 and has developed VMS support for several VAX systems. She designed and wrote the VMS CI port driver and part of the VMScluster System Communications Services. In 1988, Nancy joined the team that investigated alternatives to the VAX architecture and drafted the proposal for the Alpha AXP architecture and for porting the OpenVMS operating system to it. Nancy is a senior consulting software engineer and technical director for the OpenVMS AXP Group. She holds an A.B. degree in physics from Cornell University.

Thomas R. Benson A consulting engineer in the OpenVMS AXP Group, Tom Benson was the project leader and principal designer of the VAX MACRO-32 compiler. Prior to his Alpha AXP contributions, he led the VMS DECwindows FileView and Session Manager projects and brought the Xlib graphics library to the VMS operating system. Earlier, he supported an optimizing compiler shell used by several VAX compilers. Tom joined Digital's VAX Basic project in 1979, after receiving B.S. and M.S. degrees in computer science from Syracuse University. He has applied for four patents related to his Alpha AXP work.

Wayne M. Cardoza Wayne Cardoza is a senior consultant engineer in the OpenVMS AXP Group. Since joining Digital in 1979, he has worked in various areas of the OpenVMS kernel. Wayne was also one of the architects of PRISM, an earlier Digital RISC architecture; he holds several patents for his work. More recently, Wayne participated in the design of the Alpha AXP architecture and was a member of the initial design team for the OpenVMS port. Before coming to Digital, Wayne was employed by Bell Laboratories. Wayne received a B.S.E.E. from Southern Massachusetts University and an M.S.E.E. from MIT.

Ravindran Jagannathan Ravindran Jagannathan is a principal software engineer in the OpenVMS Performance Group currently investigating OpenVMS AXP multiprocessing performance. Since 1986, he has worked on performance analysis and characterization, and algorithm design in the areas of OpenVMS services, SMP, VAXcluster systems, and host-based volume shadowing. Ravindran received a B.E. (honors, 1983) from the University of Madras, India, and M.S. degrees (1986) in operations research and statistics and in computer and systems engineering from Rensselaer Polytechnic Institute.

Benjamin J. Thomas III Benjamin Thomas joined the OpenVMS AXP project in 1989 as project leader for I/O subsystem design and porting. In this role, he has also contributed to the I/O architecture of current and future AXP systems. Ben joined Digital in 1982 and has worked in the VMS group since 1984. In prior work, he was the director of software engineering at a microcomputer firm. Ben is a consulting engineer and has a B.S. (1978) in physics from the University of New Hampshire and an M.S.C.S. (1990) from

Worcester Polytechnic Institute.