Using Simulation to Develop and Port Software

1 Abstract

Among the tools developed to support Digital's Alpha AXP program were four software simulators. The Mannequin and ISP instruction set simulators were used to port the OpenVMS and OSF/1 operating systems to the Alpha AXP platform. The Alpha User-mode Debugging Environment (AUD) allowed Alpha AXP user-mode code to be debugged with support from the OpenVMS VAX run-time environment on VAX hardware. AUD was built from a combination of new and existing Digital software components. The Alpha User-mode Debugging Environment for Translated Images (AUDI) allowed translated images to be debugged on a simulator running on a VAX computer. With these debugging environments, user-mode applications and code components could be tested before Alpha AXP hardware and operating system software were available.

Digital developed several software simulators to support its Alpha AXP program.[1] These tools enabled engineers to develop and port software for the 64-bit RISC Alpha AXP architecture concurrently with hardware development. The simulators were used for a variety of purposes including porting, testing, verification, and performance analysis. This paper discusses four Alpha AXP software simulators: Mannequin, ISP, AUD, and AUDI.

2 The Mannequin and ISP Simulators

Two Alpha AXP instruction set simulators, Mannequin and ISP, were used to port operating systems to the Alpha AXP platform. The OpenVMS group used the Mannequin simulator to port the OpenVMS VAX system to the Alpha AXP platform. Likewise, the OSF/1 group used the ISP simulator in their port of the ULTRIX and OSF/1 operating systems to the Alpha AXP platform. Both simulators were also used for architectural and design verification, and for performance modeling.

The Mannequin simulator grew out of a simulator developed for an earlier RISC project at Digital. The ISP simulator was written anew by engineers closely associated with the Alpha AXP architecture.

The two development groups were requested to boot their respective operating systems on the simulators before attempting to boot on the Alpha Demonstration Unit (ADU), the Alpha AXP prototype hardware.[2] The simulators were so successful in tracking the Alpha AXP architecture and in rooting out software bugs that the OSF/1 group was able to boot the ULTRIX operating system on the hardware on the first attempt. The OpenVMS group had similar success and booted the OpenVMS AXP operating system in a few hours.

Note that the Alpha Demonstration Unit (ADU) is an Alpha AXP prototype hardware system and should not be confused with the Alpha User-mode Debugging Environment (AUD) or the Alpha User-mode Debugging Environment for Translated Images (AUDI), two software simulator facilities discussed later in the paper.

OpenVMS AXP Porting

The OpenVMS group used Mannequin as their Alpha AXP instruction simulator in porting the OpenVMS VAX operating system to the Alpha AXP hardware. Never before had an OpenVMS porting effort been able to debug as much operating system code in advance of hardware. Prior porting efforts debugged only up to VMB, a primary boot stage in the OpenVMS operating system. Using Mannequin, operating system developers were able to boot the entire operating system on the simulator and actually log in and debug utilities.

Some developers used Mannequin's own windows interface and debugging facilities to debug their code. Others ran the XDelta utility on top of Mannequin.[3] XDelta is a low-level system debugger used to debug the OpenVMS VAX kernel and drivers. However, the Mannequin interface was useful in initially debugging XDelta, since the Alpha AXP console allows neither breakpoints nor single stepping.

To debug their code before the full OpenVMS AXP operating system was available, other developers used Mannequin in conjunction with the Alpha primary boot (APB) code and a test harness. Mannequin was especially useful in finding alignment faults in the boot sequence, since the alignment tools are not operational until the OpenVMS AXP system is completely booted. Alignment faults occur when an attempt is made to access a unit of data located at an address that is not a multiple of the size of the data.

Microcode Speedup

One main reason the OpenVMS team was able to debug a large part of the operating system in real time was the use of specially written microcode to speed up the simulator. Mannequin is capable of running with special user-written microcode on the VAX 8800 family of machines.[4] This microcode is an addition to the normal VAX microcode for the 8800 machines; the VAX microcode remains unchanged. With microcode support, a large subset of Alpha AXP instructions is executed in microcode and attains performance comparable to native VAX instructions. The Mannequin microcode occupies 93 percent of the total 1,024 words of the user-writable control store.

Using microcode assistance greatly speeds up Mannequin execution, yielding from 350 thousand Alpha AXP instructions per CPU second (KIPS) to a peak performance of 1 million Alpha AXP instructions per CPU second (MIPS)

on a VAX 8800. Without microcode assistance, Mannequin performance is on the order of 10 KIPS. (For comparison, the ISP simulator operates at approximately 30 KIPS.) Code streams that execute completely in Mannequin microcode show much better performance than those that switch back and forth between microcode and the software simulator. With microcode

assistance on an unloaded VAX 8800, it takes from 20 to 30 minutes to boot the OpenVMS AXP system and reach the Digital Command Language (DCL) prompt after login. Because of this microcode speedup, software engineers were able to simulate and debug a much larger part of the operating system and utilities than ever before.

OSF/1 AXP Porting

The OSF/1 operating system group used the ISP simulator as an Alpha AXP instruction compute engine. The strategy was to connect the ISP simulator to dbx, a standard UNIX source-level debugger, via dbx's remote interface. An interface was added to the ISP to support the following low-level debugger commands:

- Instruction stream examine and deposit
- Data stream examine and deposit
- Register examine and deposit
- Single step
- Continue
- Boot

The dbx debugger was modified to work with the 64-bit Alpha AXP architecture. That is, addresses in the debugger were extended to 64 bits, and an Alpha AXP disassembler was provided. The ISP simulator and dbx debugger operated as separate processes communicating on the same machine by means of a socket. A socket is a protocol-independent connection point for interprocess communications.

Historically, the OSF/1 group used the ISP-dbx combination to port the ULTRIX operating system to the Alpha AXP platform as an advanced development effort. When the group began to port the OSF/1 system, Alpha AXP prototype hardware (ADUs) and field-test compilers were available. Thus, the OSF/1 group used the ISP in its ADU mode, where the ISP simulator operated as a console to the ADU hardware system. The ADU consists of an Alpha AXP DECchip 21064 processor, memory, disks, Ethernet, and a DECstation 5000 workstation, which acted as the console interface. Instructions that normally execute on the simulator were transferred to the ADU for execution. However, the entire symbolic debugging environment remained unchanged.

Simulator Specifics

The ISP simulator was written entirely in portable C. The Mannequin simulator was a hybrid of the C++ and C languages. ISP consisted of approximately 25,000 lines of code, Mannequin 31,800 lines. The two simulators shared common code: the ISP simulator provided Mannequin with floating-point routines and a comprehensive instruction test program; Mannequin provided ISP with I/O device routines. Thus, the simulators verified the Alpha AXP architecture as well as each other.

The Mannequin and ISP simulators tracked and supported changes in the evolving Alpha AXP architecture and in PALcode. PALcode is special machine-dependent software that provides support for many low-level operating system services such as faults and exceptions. PALcode also provides instructions not in the base Alpha AXP hardware.

The two simulators have features common to many simulators, including support for loading and running programs, setting breakpoints and watchpoints, accessing memory, and saving and restoring machine state. Also supported are many machine-specific features, such as I/O devices, interval timers, and configurable translation lookaside buffers. Besides a command line interface, the Mannequin simulator has a graphical windows interface that allowed users to see most machine resources in a windows-based format, as shown in Figure 1.

The Mannequin and ISP simulators support three basic devices:

- A console device used for terminal I/O
- A disk device used to boot the operating system
- An interval timer used for interrupts

The disk device on the simulators can be either a file or a physical disk device. The OpenVMS group used a shared disk so that developers could boot from a common disk while running on the simulator.

The simulators provide 16 megabytes (MB) of physical memory with a default page size of 8 kilobytes (kB). The physical memory of the simulators may be increased to the practical limit of available virtual memory on a VAX system (minus a small amount for the actual simulator code).

Both simulators have configurable instruction stream (I-stream) and data stream (D-stream) translation lookaside buffers (TLBs). A TLB is a small cache that holds recent virtual-to-physical address translation and protection information. The simulator TLBs can have a variable number of entries in each of the four granularity hint block sizes. Granularity hints indicate to the translation buffer implementations that a block of pages can be treated as a single, larger page. In essence, there are four minitranslation buffers. The ISP simulator supports selectable TLB replacement algorithms, whereas Mannequin supports only the not-last-used (NLU) algorithm. The configurable TLBs allowed the operating system and chip design groups to analyze and finely tune the translation lookaside buffers for optimum performance.

Performance Analysis and Benchmarking

The Mannequin and ISP simulators also support execution of user-mode, stand-alone programs, i.e., those with little or no operating system run-time support, by providing program loaders for several formats. These formats include two UNIX object formats (COFF and a.out), an OpenVMS AXP image format, and a system (raw data) image format.

Programs were compiled with early field-test Alpha AXP compilers. Program execution was especially useful for hardware designers and compiler writers for performance analysis and benchmarking purposes. Note that applications requiring full operating system support used the AUD facility, described in a later section.

The simulators can generate trace files in a standard trace file format. This commonality enabled the two facilities to share the same trace analysis tools. The trace files generated by Mannequin and ISP were also used as input to the Alpha Performance Model, another simulator that generated detailed performance data.

EVILIST and ALPHA$REPORT were two tools frequently used to analyze trace files and generate statistics concerning machine resources used during program execution. The types of data generated by ALPHA$REPORT include the following:

- Instruction distribution by opcode, class, and format
- Instruction and floating-point register utilization summary
- Distribution of code block run lengths
- Opcode pair distribution by class
- Control/branch instruction flow summary

An actual trace analysis report generated by ALPHA$REPORT is shown in Figure 2. This example comes from a scaled version of FPPPP (one of the 14 benchmarks in the SPECfp92 floating-point test suite), with the constant NATOMS set equal to 2. Figure 2 displays a report listing instruction distribution by opcode.

Alpha AXP operating system developers and compiler writers relied heavily on the trace reports for help in designing critical sections of code. For example, the register usage distribution report helped determine how many registers should be preserved by a call and how many should be scratch (usable by a called routine without being preserved).

3 The AUD Facility

Whereas the Mannequin and ISP simulators were suitable for initial debugging of low-level software such as operating systems, direct use of these tools for user-mode applications, i.e., layered products, is a different matter. Porting and debugging Alpha AXP user-mode code is at best difficult without the full run-time support of an operating system. User-mode applications typically take advantage of a wide variety of run-time libraries, including compiled code support (such as the Fortran run-time library), mathematical routines, graphics I/O services, and database software (such as Rdb for OpenVMS). Even if all this software were immediately available for Alpha AXP systems, running it under simulation would be prohibitively slow.

Therefore, Digital developed a mixed-execution debugging environment. This Alpha User-mode Debugging Environment (AUD) was built from a combination of new and existing Digital software components. In the AUD environment, user-mode code being developed for or ported to the Alpha AXP platform could be compiled and executed as Alpha AXP code using simulation on VAX hardware. At the same time, OpenVMS VAX run-time services called by the code could be executed as native VAX instructions. Thus, modules could be ported and debugged one at a time, until almost the entire application consisted of bug-free Alpha AXP code.

During the design of the AUD environment, two key technical issues were

- How to efficiently detect calls made by executing VAX code to a routine in Alpha AXP code that could be "executed" only by simulation, and conversely, how to detect calls made by Alpha AXP code being simulated to native VAX code.

- How to effect the transformation of parameters, both location and representation, from that provided by the caller in one domain into the locations and representations expected by the called routine in the other domain. Although there existed well-defined and widely followed calling standards for both domains, a variety of special-purpose, high-performance calling conventions were used in many situations.

This mixed-execution environment was expected to have a relatively short lifetime, because it would become obsolete as soon as significant numbers of real Alpha AXP hardware systems became available. Consequently, AUD itself had to be simple and inexpensive enough to be created quickly and put into use. The development effort met this requirement. The elapsed time from initial concept to first use was about eight months; the total development effort for AUD over its lifetime was between three and four man-years.

AUD Components

Despite the desire for simplicity, AUD consists of a number of cooperating components:

- Callable Mannequin Alpha Simulator
- AUD debugger
- AUD linker
- Alpha AXP native services
- VAX jacketing services
- AUD Linkage Analyzer (ALA)
- Selected VAX jackets

Callable Mannequin Alpha Simulator. Callable Mannequin, the Alpha AXP instruction set simulator, is essentially a subset of the Mannequin simulator described earlier. In particular, Callable Mannequin omits the user interface and Alpha AXP machine state. Instead, the AUD debugger supplies the user interface. Also, storage for the Alpha AXP machine state is separately linked into the AUD environment to make this information globally accessible. Callable Mannequin does retain the microcode-assist feature.

AUD Debugger. The AUD debugger is a modified version of DEBUG-32, the user-mode debug utility on the OpenVMS VAX operating system. The AUD debugger provides most of the same features of DEBUG-32. A configuration option allows the DEBUG-32 utility to use an internal, low-level remote debugger interface to interface with a foreign target. (This capability was originally developed for use in other products such as VAXELN Ada.) We developed new code to join DEBUG-32 and Mannequin using this interface. As a result, the AUD debugger works directly with VAX code, in the usual manner, and works with Alpha AXP code by passing commands to the Callable Mannequin simulator to set breakpoints, examine instructions, execute code, etc.

AUD Linker. The AUD linker is a variant of the Alpha AXP cross linker that reads Alpha AXP object modules as input and produces an OpenVMS VAX format image as output. The standard VAX linker can therefore reference locations in the Alpha AXP image in the normal way, and the standard OpenVMS image activator can be used to load the Alpha AXP image for execution. However, to minimize complexity, we did constrain the Alpha AXP image to be linked as an absolute image (i.e., a based image, in

OpenVMS jargon). This restriction eliminated the problem of how to relocate Alpha AXP instructions using the OpenVMS image activator. As mentioned previously, the Alpha AXP image also includes a global storage area to hold the simulated Alpha AXP machine state.

Alpha AXP Native Services. Alpha AXP native services is a special operating system shell, part of which executes as Alpha AXP code (under simulation) and part of which is included in the AUD jacketing services. The native services provide the lowest-level support for hardware exception handling and the OpenVMS condition-handling facility. While AUD ultimately supported frame-based condition handling within the Alpha AXP image, interoperation of application exceptions between the Alpha AXP and VAX domains was not supported.

VAX Jacketing Services. VAX jacketing services is VAX code that supports the ability to write jackets that pass control back and forth between VAX and Alpha AXP code. The mechanics for accomplishing this are discussed in the Jacketing section.

AUD Linkage Analyzer. The ALA is a specialized compiler that reads a specialized jacket description language. This language describes how calls in one domain are to be transformed into calls in the other domain on a routine-by-routine, parameter-by-parameter basis. The output from the ALA is an Alpha AXP object module and a linker options control file, both used to link the Alpha AXP image, and a VAX object module. The Alpha AXP object module provides a transfer vector into the Alpha AXP procedures. The linker options control file provides symbol definitions in an encoded form to manage calls from the Alpha AXP image to the main VAX image, which is linked later. The VAX object module contains a table that encodes the jacketing description.

Selected VAX Jackets. Selected VAX jackets are ALA jacketing files (in both source and compiled forms) for calling common VAX facilities from Alpha AXP code. Jackets are provided for OpenVMS system services, the C run-time library, and some parts of the general-purpose, run-time library (LIBRTL). The DECwindows group also supplied jacket definition files for use by other groups. AUD users are able to supplement these files as needed by creating and compiling their own jacketing descriptions for other VAX facilities.

Figure 3 shows the main steps in building an AUD environment. The uppermost sequence shows the compilation and linking of the Alpha AXP components, which results in the creation of the Alpha AXP image. The central sequence shows the compilation of the jacket descriptions, which results in the creation of components that are included in both the Alpha AXP and the VAX images. The lower rows of Figure 3 show the compilation of the VAX part of an application and its linking with the AUD manager to create the VAX main image. When the mixed VAX and Alpha AXP application is executed, these images are combined in memory with Callable Mannequin, the AUD debugger, and other shareable images. This relationship is illustrated in Figure 4.

Jacketing

Jacketing is the key feature that allows VAX and Alpha AXP interoperability, i.e., gives a processor the appearance of being able to execute both VAX and Alpha AXP instructions. Although the details of jacketing are intricate, the result is simple and elegant. Calls can be made freely back and forth between VAX compiled code and Alpha AXP compiled code, without any special compilation modes on either side. The AUD support is fully recursive and reentrant.

Static calls from VAX to Alpha AXP code are directed to dummy entry points in the object module produced by the ALA compiler. Each entry point is simply an instruction that loads a pointer to the jacketing description table for the target Alpha AXP procedure, followed by a transfer into common jacket interpretation code.

Calls from Alpha AXP code to VAX code use the fact that the Callable Mannequin component stops and returns control to the AUD environment when it detects an instruction that transfers control out of the Alpha AXP image. In this case, the apparent address is an encoded integer (created by the ALA), whose high four bits make it look like an illegal address (in the VAX reserved S1 space) and whose remaining bits are a two-level index (i.e., 12 bits of facility code and 16 bits of offset) into the jacket description table for the target VAX procedure. This two-level scheme was chosen to allow jacket descriptions for different shared library facilities to be prepared and compiled independently. The facility code is a number normally already associated with that facility by software convention for other purposes.

When a routine is called using a dynamically determined address, such as an address given in a function variable, a property of the VAX and Alpha AXP architectures is exploited to determine dynamically whether the target routine is a VAX routine or an Alpha AXP routine. According to the VAX architecture, the first 16 bits of a routine comprise a mask that encodes the registers to be preserved as part of the call. Bits 12 and 13 of this mask are unused and required to be 0; if one of these bits is set at the time of a call, then a hardware exception results. According to the OpenVMS AXP software architecture, an Alpha AXP procedure address is actually the address of a procedure descriptor, which is a data structure and not the actual Alpha AXP code. By design, bits 12 and 13 of this data structure must be set to 1.

VAX execution of a VAX CALL instruction that attempts to transfer to an Alpha AXP procedure results in an exception. A special AUD exception handler intercepts the exception, determines if the illegal entry mask is caused by a reference into an Alpha AXP image, and if so, calls into the AUD jacketing routines to reformat the call frame. This mechanism also

works for handling asynchronous system traps (ASTs) from the OpenVMS VAX operating system to Alpha AXP code.

For computed calls from Alpha AXP code, compiled code calls an Alpha AXP run-time library routine to perform the comparable bit 13 test (under simulation). If bit 13 of the target location is set to 1, then simulated execution continues and an Alpha-to-Alpha call is carried out. Otherwise, control transfers to a special VAX code entry point in AUD, which terminates simulation and performs jacketing back to the VAX target procedure.

Basic Operation

To start executing a mixed application, the AUD environment first performs several initialization steps. In particular, AUD scans all the images loaded in process memory to identify the Alpha AXP image (only one was allowed and supported).

Some AUD options are set through the use of OpenVMS logical names, which are interrogated during image initialization. These options include

- Selecting Alpha AXP stack size
- Enabling delivery of ASTs to Alpha AXP routines
- Disabling the normal Alpha AXP stack consistency checks
- Disabling unaligned memory reference messages
- Enabling AUD initialization tracing
- Disabling integer overflow checking

Debugging combined VAX and Alpha AXP code under the AUD environment is similar to debugging normal VAX code under the DEBUG-32 OpenVMS debugger. Basically, if the address involved in a debug command is within an Alpha AXP image, then the debugger calls the Mannequin simulator to perform the command for the Alpha AXP code. Otherwise, the DEBUG-32 debugger itself performs the command for the VAX code, as usual. Alpha AXP machine state is kept in static global storage by Mannequin and thus is visible to the AUD debugger.

In the DEBUG symbol table (DST) representation, variables that are allocated in the Alpha AXP registers are described as being allocated in the corresponding global state locations. This "trick" allowed AUD to handle the 64 Alpha AXP registers using the VAX DST representation, which can encode only the 16 VAX registers.

Once simulation begins, Mannequin continues to simulate Alpha AXP instructions until it either detects an instruction that would transfer

control outside of the Alpha AXP image, completes a single-step request, or detects an error condition. Upon returning to the AUD environment, Mannequin supplies status information that indicates the reason simulation ended.

For a transfer of control from Alpha AXP to VAX code, AUD must first determine whether the transfer is a return from Alpha AXP code as a result of a prior VAX call or a new call from Alpha AXP code to VAX code. AUD is fully reentrant, so AUD cannot make this determination from global state. If the target address is a distinguished address that AUD supplies when it sets up a VAX-to-Alpha call (i.e., an address in the reserved S1 part of the VAX address space), the address is interpreted as a return transfer. Otherwise, AUD initiates a new Alpha-to-VAX call.

For a return operation, the AUD code copies the return value or values from the Alpha AXP registers and passes them back to the VAX code. A VAX return instruction is then executed to resume execution of the calling VAX code.

For a call operation, the VAX code fetches the Alpha AXP parameters and builds a VAX argument list, which is then used to call the target VAX routine. When the VAX routine returns, the contents of the result registers are copied to the corresponding Alpha AXP machine state locations, and Mannequin is restarted to resume executing Alpha AXP code.

Despite some limitations (e.g., only one Alpha image and no exception handling across the VAX to Alpha AXP domains), AUD greatly aided the OpenVMS AXP porting effort. The simulator provided software groups with a pseudo-Alpha AXP environment in which to debug their Alpha AXP code, well before either Alpha AXP hardware or the OpenVMS AXP operating system was available. Many OpenVMS AXP groups successfully used AUD to facilitate their porting, including the Record Management Services (RMS), DECwindows, Forms Management System (FMS), various OpenVMS command utilities, text processing utility (TPU), DEBUG, and GEM compiler back-end groups.

4 The AUDI Facility

The VAX Environment Software Translator (VEST) is an important part of the initial OpenVMS AXP offering.[5] VEST translates an OpenVMS VAX executable or shareable image into an OpenVMS AXP image that can then be executed with support on an OpenVMS AXP system. As for other user-mode layer software components, it was desirable to test VEST and images translated by VEST as early as possible in a simulation environment such as AUD. However, AUD could not be used directly to test translated images for two reasons:

- VEST directly creates an Alpha AXP image. In effect, VEST is a combined compiler and linker. Thus, the symbol mapping protocols used by AUD were extraneous, and the linking protocols had to be completely replaced.

- Actual execution of a translated image on an OpenVMS AXP system makes use of the Translated Image Environment (TIE).[5] The TIE is a shareable library that contains support routines for translated images. In particular, TIE provides support for VAX complex instruction

processing, VAX-to-Alpha address mapping, and OpenVMS VAX exception handling. Creating a VAX version of the TIE to use with AUD required intimate interfaces with the OpenVMS VAX operating system as well as compatibility with AUD.

Thus, the need to debug translated images led to the creation of the Alpha User-mode Debugging Environment for Translated Images (AUDI). Just as Callable Mannequin provided a key building block for AUD, AUD in turn provided a key building block for AUDI. Alpha AXP software teams and porting centers used AUDI to port both Digital and third-party translated applications prior to the arrival of Alpha AXP hardware. The porting process was as follows: a VAX application was translated to Alpha AXP code by means of the VEST translator; this code was then run on a VAX system using the AUDI simulator.

The AUDI process components shown in Figure 5 include the

- Callable Mannequin Alpha simulator
- AUD debugger
- VAX version of the TIE
- Translated VAX code (Alpha AXP code)

AUDI Environment

Emulated VAX state in AUDI is maintained in a global context block. Emulated VAX registers R0 through R14 are used exactly as their VAX counterparts. The correspondence between a translated and equivalent VAX program counter (PC) is not directly available during execution, since translated code occupies different address space than the original VAX code. Thus, register R15 is used instead as an in-image index register.

The user-mode VAX stack is split into a VAX stack and an Alpha and emulated VAX stack. The VAX stack services both the AUDI environment and any VAX system services or run-time library routines that the translated image may call. The Alpha and emulated VAX stack services Alpha AXP and translated code.

Translated images contain calls to the TIE as necessary to simulate VAX complex instructions and procedure calls. Complex instruction routines are used to simulate VAX instructions that would otherwise expand into excessive Alpha AXP code. However, since AUDI is running on VAX hardware, complex instructions can be executed native on the VAX hardware.

To initialize the AUDI environment, the translated image calls an initialization routine in the TIE by means of an initialization program section (PSECT). This routine determines the address range of the Alpha AXP code and the location of the VAX-to-Alpha address mapping structure, saves the current Alpha AXP register state, and calls Mannequin to begin executing translated code at the appropriate entry point. Translated code uses the address mapping structure to find computed branch destinations on the fly. Callable Mannequin then executes translated code until it encounters some instruction that would transfer control out of translated code. The cause of this transfer would be either a TIE-based procedure or complex instruction call, or calls to native VAX routines.

Like AUD, AUDI allows interoperation between translated VAX code (Alpha AXP code) and VAX code. Translated code can use existing VAX system services and run-time libraries. AUDI does not use the jacketing language described in the section The AUD Facility. Instead, AUDI automatically jackets procedure calls between the translated VAX code and the native VAX code. Autojacketing includes building proper parameter lists and call frames for the destination calling standard.

The fact that AUDI does not use a jacketing language leads to some procedure call limitations. However, note that these limitations do not appear when running translated code on actual Alpha AXP hardware. For incoming calls (VAX code to translated VAX code), all AST delivery and condition handlers execute as VAX code rather than as translated VAX code. Thus, translated programs may not function properly. For outgoing

calls (translated VAX code to VAX code), routines in which a callee modifies its caller's stack frame argument list or return address may produce unpredictablte results, since the autojacketing may be altered or disconnected.

AUDI Example

Figure 6 shows the execution of a translated image under AUDI. Note that both the BASIC image (HELLO_WORLD) and the BASIC run-time library (BASRTL) are translated. Run-time libraries that are used by the AUDI environment cannot be translated under AUDI. Translating run-time libraries that AUDI itself uses causes a "circularity in activation" and incorrect or no execution.

In the HELLO_WORLD example, there are 28 calls to VAX routines, most likely those to LIBRTL and OpenVMS system services. There are 21 unique CALLx contexts and 7 reused ones. In addition, the example uses four different complex instructions.

5 Summary

The software simulators Mannequin, ISP, AUD, and AUDI greatly aided Alpha AXP software porting and development efforts. Substantial parts of both system and application software were simulated and verified concurrently with hardware development. When Alpha AXP hardware became available, most software could be plugged in simply and ran exactly as expected. The use of these simulation tools saved a year or more from the overall Alpha AXP schedule.

6 Acknowledgments

Many people throughout Digital contributed to the success of the Alpha AXP simulators. Homayoon Akhiani, Ray Lanza, Stephan Meier, Steve Morris, Andrew Payne, and Jon Reeves worked on the ISP model. George Darcy, Mark Herdeg, Kevin Koch, Eric Rasmussen, and Scott Robinson contributed to the Mannequin simulator. The AUD effort included several groups from across Digital. Their primary contributors were Walter Arbo, Ronald Brender, Henry Grieb, Mark Herdeg, Michael Iles, James Johnson, Robert Landau, Maurice Marks, Dennis Murphy, Scott Robinson, Larry Woodman, and James Wooldridge. Finally, much of the AUDI information in this article is taken from work originally done by Scott Robinson. Other AUDI contributors include George Darcy, Mark Herdeg, Matthew Kirk, Naghmeh Mirghafori, and Murari Srinivasan.

7 References

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

2. C. Thacker, D. Conroy, and L. Stewart, "The Alpha Demonstration Unit: A High-performance Multiprocessor for Software and Chip Development," Digital Technical Journal, vol. 4, no. 4 (1992, this issue): 51-65.

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

4. S. Mishra, "The VAX 8800 Microarchitecture," Digital Technical Journal, vol. 1, no. 4 (February 1987): 20-33.

5. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," Digital Technical Journal, vol. 4, no. 4 (1992, this issue): 137-152.

8 Trademarks

The following are trademarks of Digital Equipment Corporation:

Alpha AXP, AXP, DEC, DECchip 21064, DECstation, DECwindows, Digital, OpenVMS, OpenVMS AXP, OpenVMS VAX, DEC Rdb for OpenVMS, ULTRIX, VAX, VAX 8800, DECstation, and DECwindows.

The following are third-party trademarks:

OSF/1 is a registered trademark of Open Software Foundation, Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

9 Biographies

George A. Darcy III As a senior software engineer in the Alpha Migration Tools Group, George Darcy has worked on the Mannequin Alpha AXP simulator, the VEST binary translator, and the Translated Image Environment (TIE) run-time library. In his ten years at Digital, he has also developed a virtual disk driver for the OpenVMS V5.0 SMP operating system, software behavioral models of a high-end VAX processor, and various simulation and CAD software tools. George received a B.S.C.E. (cum laude, 1984) from Boston University, where he was an Engineering Merit Scholar and a member of Tau Beta Pi.

Ronald F. Brender Ron Brender is a senior consultant software engineer, contributing to the GEM compiler back-end project in the Software Development Technologies Group. He has worked on compilers and programming language definition for Alpha AXP, VAX, PDP-11, and PDP-10 systems, including Ada, FORTRAN and BLISS. A member of various standards committees since the mid-1970s, Ron is now responsible for VAX and Alpha AXP calling standards. He joined Digital in 1970, after receiving a Ph.D. in computer and communication sciences at the University of Michigan.

Stephen J. Morris Stephen Morris is a consultant software engineer in the Semiconductor Engineering Advanced Development Group. In addition to writing the Alpha ISP simulator, he wrote the OpenVMS and OSF PALcode for the Alpha AXP program. In previous work, Stephen designed the control sections of the instruction prefetch and translation look-aside buffer for an experimental Digital RISC chip. He also worked on the MicroVAX chip team, doing console and debug work, and in the RSTS/E operating system group. Stephen joined Digital after receiving a B.A. in biology from the University of Rochester in 1977.

Michael V. Iles Michael Iles is a senior technology consultant at the UK Alpha AXP Migration Centre. Since joining Digital in 1975, Mike has worked in various field positions, in Advanced VAX development as a microcoder,

and for VMS engineering as a software engineer. He worked on the migration of OpenVMS VAX to the Alpha AXP platform, designing and implementing a user-mode simulation environment that became AUD. Mike has a B.Sc. in electrical engineering (honors, 1973) from City University, London, and

holds a patent for digital speech synthesis techniques. He has several patents pending for AUD.

Copyright 1992 Digital Equipment Corporation. Forwarding and copying of this article is permitted for personal and educational purposes without fee provided that Digital Equipment Corporation's copyright is retained with the article and that the content is not modified. This article is not to be distributed for commercial advantage. Abstracting with credit of Digital Equipment Corporation's authorship is permitted. All rights reserved.