Binary Translation

1 Abstract

Binary translation is a technique used to change an executable program for one computer architecture and operating system into an executable program for a different computer architecture and operating system. Two binary translators are among the migration tools available for Alpha AXP computers: VEST translates OpenVMS VAX binary images to OpenVMS AXP images; mx translates ULTRIX MIPS images to DEC OSF/1 AXP images. In both cases, translated code usually runs on Alpha AXP computers as fast or faster than the original code runs on the original architecture. In contrast to other migration efforts in the industry, the VAX translator reproduces subtle CISC behavior on a RISC machine, and both open-ended translators provide good performance on dynamically modified programs. Alpha AXP binary translators are important migration tools - hundreds of translated OpenVMS VAX and ULTRIX MIPS images currently run on Alpha AXP systems.

When Digital started to design the Alpha AXP architecture in the fall of 1988, the Alpha AXP team was concerned about how to run existing VAX code and soon-to-exist MIPS code on the new Alpha AXP computers.[1,2] To take full advantage of the performance capability of a new computer architecture, an application must be ported by rebuilding, using native compilers. For a single program written in a standard programming language, this is a matter of recompile and run. A complex software application, however, can be built from hundreds of source pieces using dozens of tools. A native port of such an application is possible only when all parts of the build path are running on the new architecture.

Therefore, devising a way to run an existing (old architecture) binary version of a complex application on a new architecture is an important interim measure. Such a technique allows a user to get applications up and running immediately, with minimal porting effort. Once a user's everyday environment is established, applications can be rebuilt over time, using hand-written native code or partially native and partially old code.

2 Background

Several techniques are used in the industry to run the binary code of an old architecture on a new architecture. Figure 1 shows four common techniques, from slowest to fastest:

- Software interpreter (e.g., Insignia Solutions' SoftPC)
- Microcoded emulator (e.g., PDP-11 compatibility mode in early VAX computers)

o Binary translator (e.g., Hunter System's XDOS)

Binary Translation

o Native compiler

A software interpreter is a program that reads instructions of the old architecture one at a time, performing each operation in turn on a software-maintained version of the old architecture's state. Interpreters are not very fast, but they run on a wide variety of machines and can faithfully reproduce the behavior of self-modifying programs, programs that branch to data, programs that branch to a checksum of themselves, etc. Caching interpreters gain speed by retaining predecoded forms of previously interpreted instructions.

A microcoded emulator operates similarly to a software interpreter but usually with some key hardware assists to decode the old instructions quickly and to hold hardware state information in registers of the micromachine. An emulator is typically faster than an interpreter but can run only on a specific microcoded new machine. This technique cannot be used to run existing code on a reduced instruction set computer (RISC) machine, since RISC architectures do not have a microcoded hardware layer underlying the visible machine architecture.

A translated binary program is a sequence of new-architecture instructions that reproduce the behavior of an old-architecture program. Typically, much of the state information of the old machine is kept in registers in the new machine. Translated code faithfully reproduces the calling standard, implicit state, instruction side effects, branching flow, and other artifacts of the old machine. Translated programs can be much faster than interpreters or emulators, but slower than native-compiled programs.

Translators can be classified as either (1) bounded translation systems, in which all the instructions of the old program must exist at translate time and must be found and translated to new instructions,[3,4,5] or (2) open-ended translation systems, in which code may also be discovered, created, or modified at execution time. Bounded systems usually require manual intervention to find 100 percent of the code; open-ended systems can be fully automatic.

To run existing VAX and MIPS programs, an open-ended system is absolutely necessary. For example, some customer programs write license-check code (VAX instructions) to memory, and branch to that code. A bounded system fails on such programs.

A native-compiled program is a sequence of new-architecture instructions produced by recompiling the program. Native-compiled programs usually use newer, faster calling conventions than old programs. With a well-tuned optimizing compiler, native-compiled programs can be substantially faster than any of the other choices.

Most large programs are not self-contained; they call library routines, windowing services, databases, and toolkits, for example. These programs also directly or indirectly invoke operating system services. In simple environments with a single dominant library, it can be sufficient to rewrite that library in native code and to interpret user programs,

particularly user programs that actually spend most of their time in the library. This strategy is commonly used to run Windows and Macintosh programs under the UNIX operating system.

In more robust environments, it is not practical to rewrite all the shared libraries by hand; collections of dozens or even hundreds of images (such as typical VAX ALL-IN-1 systems) must be run in the old environment, with an occasional excursion into the native operating system. Over time, it is desirable to rebuild some images using a native compiler while retaining other images as translated code, and to achieve interoperability between these old and new images. The interface between an old environment and a new one typically consists of "jacket" routines that receive a call using old conventions and data structures, reformat the parameters, perform a native call using new conventions and data structures, reformat the result, and return.

The Alpha AXP Migration Tools team considered running old VAX binary programs on Alpha AXP computers using a simple software interpreter, but rejected this method because the performance would be too slow to be useful. We also rejected the idea of using some form of microcoded emulator. This technique would compromise the performance of a native Alpha AXP implementation, and VAX compatibility would be nearly impossible to achieve without microcode, which is inconsistent with a high-speed RISC design.

We therefore turned to open-ended binary translation. We were aware of the earlier Hewlett-Packard binary translator, but its single-image HP 3000 input code looked much simpler to translate than large collections of hand-coded VAX assembly language programs.[6] One member of the team (R. Sites) wrote a VAX-to-VAX binary translator in October 1988 as proof-of-concept. The concept looked feasible, so we set the following ambitious product goals:

1. Open-ended (completely automatic) translation of almost all user-mode applications from the OpenVMS VAX system to the OpenVMS AXP system

2. Open-ended translation of almost all user-mode applications from the ULTRIX system to the DEC OSF/1 system

3. Run-time performance of translated code on Alpha AXP computers that meets or exceeds the performance of the original code on the original architecture

4. Optional reproduction of subtle old-architecture details, at the cost of run-time performance, e.g., complex instruction set computer (CISC) instruction atomicity for multithreaded applications and exact arithmetic traps for sophisticated error handlers

5. If translation is not possible, generation of explicit messages that give reasons and specify what source changes are necessary

While we were creating the VAX translator, we discovered that the process of building flow graphs of the code and tracking data dependencies yielded information about source code bugs, performance bottlenecks, and dependencies on features not available in all Alpha AXP operating systems. This analysis information could be valuable to a source code maintainer. Thus, we added one more product goal:

6. Optional source analysis information

To achieve these goals, the Alpha AXP Migration Tools team created two binary translators: VEST, which translates OpenVMS VAX binary images to OpenVMS AXP images, and mx, which translates ULTRIX MIPS images to DEC OSF/1 AXP images. However, binary translation is only half the migration process. As shown in Figure 2, the other half is to build a run-time environment in which to execute the translated code. This second half of the process must bridge any differences between old and new operating systems, calling standards, exception handling, etc. For open-ended translation, this part of the process must also include a way to run old code that was not discovered (or did not exist) at translate time. The translated image environment (TIE) and mxr run-time environment support the VEST and mx translators, respectively, by reproducing the old operating environments. Each environment supports open-ended translation by including a fallback interpreter of old code, and extensive run-time feedback to avoid using the interpreter except for dynamically created code. Our design philosophy is to do everything feasible to stay out of the interpreter, rather than to increase the speed of the interpreter. This approach gives better performance over a wider range of programs than using pure interpreters or bounded translation systems.

The reminder of this paper discusses the two binary translator/run-time environment pairs available for Alpha AXP computers: VEST/TIE and mx/mxr. To establish a basis for the discussion, the reader must understand the following terms: datum, alignment, instruction atomicity, granularity, interlocked update, and word tearing. Definitions of these terms appear in the References and Note section.[7]

3 VEST: Translating a VAX Image

Translating a VAX image involves two main steps: analyzing VAX code and generating Alpha AXP code. The translated images produced are OpenVMS AXP images and may be run just like native images.[8] Translated images run with the assistance of the translated image environment, which is discussed later in this paper. The VEST binary translator is written in C++ and runs on VAX, MIPS, and Alpha AXP machines. The TIE is written in the OpenVMS system programming languages, BLISS and Alpha assembler.

Analysis

To locate VAX code, VEST starts disassembling code at known entry points and recursively traces the program's flow of control. Entry points come from main and global routines, debug symbol table entries, and optional information files (including run-time feedback from the TIE).

As VEST traces the program, it builds a flow graph that consists of basic blocks (i.e., straight-line code sequences) annotated with information derived from parsing instructions. VEST then performs several analyses on the flow graph to propagate context information to each basic block and eliminate unnecessary operations. Context information includes condition code usage, register contents, stack depth, and a variety of other information that allows VEST to generate optimized code.

Analysis is important for achieving good performance. For example, no condition codes exist in the Alpha AXP architecture. Without analysis it would be necessary to compute condition codes for each VAX instruction even if the codes were not used. Furthermore, several forms of analysis were invented to allow correct translation. For example, VEST automatically determines if a subroutine does a normal return.

Code analysis can detect many problems, including some that indicate latent bugs in the source image. VEST can detect, for example, uninitialized variables, improperly formed VAX CASE instructions, stack depth mismatches along two different paths to the same code (the program expects data to be at a certain stack depth), improperly formed returns from subroutines, and modifications to a VAX call frame. A latent bug in the source image should be fixed, since the translated image may demonstrate incorrect behavior due to that bug.

Analysis also detects the use of unsupported OpenVMS features including unsupported system services. The source image must be modified to eliminate the use of these features.

Some problems reported by VEST result from code that is hackish in nature. For example, we found code that expects a call mask at an entry point to be executed as a no-op instruction so that the code preceding the subroutine can simply execute the call mask, rather than go through the overhead of a VAX jump (JMP) instruction. VEST reproduces the behavior of the VAX program, even if this behavior is a result of luck.

A VEST-generated flow graph is displayed in Figure 3. Dashed lines represent code paths followed if a conditional branch is taken. Solid lines indicate fall-through paths. A problem is highlighted by a wide, dashed pointer whose bottom end indicates the basic block in which the problem was uncovered. Full blocks show the path that reveals the error; empty

blocks show basic blocks that are not in the error path. In Figure 3, a path exists by which register 3 (R3) may be used without being set if the VAX BNEQ (branch if the register does not equal zero) instruction in the second basic block is true the first time through the code sequence.

NOTE

Figure 3 (VEST-generated Flow Graph Showing Uninitialized Variable) is unavailable.

Code Generation

The VEST translator generates code by converting each VAX instruction into zero or more Alpha AXP instructions. The architecture mapping is straightforward because there are more Alpha AXP registers than VAX registers. The VAX architecture has only 15 registers, which are used for both floating-point and integer operations. The Alpha AXP architecture has separate integer and floating-point registers. VAX R0 through R14 are mapped to Alpha AXP R0 through R14 for all operations except floating point. R12, R13, and R14 retain their VAX designations as argument pointer, frame pointer, and stack pointer, and R15 is used to resolve PC-relative references. Floating-point operations are mapped to F0 through F14.

The VAX architecture has condition codes that may be referenced explicitly. In translated images, condition codes are mapped into R22 and R23. Similar to the HP 3000 translator, R23 is used as a fast condition code register for positive/negative/zero results.[6] R22 contains all four condition code bits and is calculated only when necessary. All remaining Alpha AXP registers are used as scratch registers or for OpenVMS AXP standard calls.

VEST connects simple branches directly to their translated targets. VEST performs backward symbolic execution of VAX instructions to resolve as many computed branch targets as feasible. If more than one possible computed target exists, a run-time lookup is done on the VAX target address. If the lookup fails to find a translated target, a fallback VAX interpreter is used, as described in the TIE section Failure to Find All Code during Translation. Unlike bounded translation systems, which must achieve 100 percent resolution of computed targets, the VEST and mx binary translators require no manual intervention.

Translated Images

A translated image has the same format as an OpenVMS AXP image and contains the original OpenVMS VAX image as well as the Alpha AXP instructions that were generated for the VAX code. The run-time VAX interpreter TIE needs the original VAX instructions as a fallback. (Also, some error handlers look up the call stack for pointers to specific VAX instructions.) The addresses of statically allocated data in the translated image are identical to their VAX addresses. The image contains a VAX-to-Alpha AXP address mapping table for use during lookups and may contain an instruction atomicity table, described in the VAX Instruction Guarantees section.

Translated images use the OpenVMS VAX calling standard. Native images use different conventions, but translated images interoperate with native or translated shareable images. Automatic jacketing services are provided in the TIE to convert calls using one set of conventions into the other. In many cases, jacketing services permit substitution of a native shareable

image for a translated shareable image without modification. However, a jacket routine is sometimes required. For example, on OpenVMS AXP systems, the translated FORTRAN run-time library, FORRTL_TV, invokes the native Alpha AXP library DEC$FORRTL for I/O-related subroutine calls. DEC$FORRTL has a different interface than FORRTL has on an OpenVMS VAX system. For these calls, FORRTL_TV contains hand-written jacket routines.

Files Used

Translating an image requires only one file - a VAX executable image. Several optional files make translation more effective.

1. Image information files (IIFs). VEST automatically creates IIFs to provide information about shareable image interfaces. The information includes the addresses of entry points, names of routines, and resource utilization.

2. Symbol information files (SIFs). VEST automatically generates SIFs to control the global symbol table in a translated shared library, facilitating interoperation between translated and native images.

3. Hand-edited information files (HIFs). The TIE automatically generates HIFs, which may be hand-edited to supply information that VEST cannot deduce. HIFs contain directives to tell VEST about undetected entry points, to force it to change specific assumptions about an image during translation, and to provide known interface properties to be propagated into an IIF.

4 VEST Performance Considerations

In evaluating translated code performance, we recognized that there was a significant trade-off between performance and the accuracy of emulating the VAX architecture. VEST permits users to select several architectural assumptions and optimizations, including:

- D-float precision. The Alpha AXP architecture provides hardware support for D-float with only 53-bit mantissas, whereas the VAX architecture provides 56-bit mantissas. The user may select translation with either 53-bit hardware support (faster) or 56-bit software support (slower).

- Alignment. Alpha AXP instructions support only naturally aligned longword (32-bit) and quadword (64-bit) memory operations. Unaligned memory operations cause alignment faults, which are handled transparently by software at significant run-time expense. The user may direct VEST to assume that data references are unaligned whenever alignment information is unavailable.

- Instruction atomicity. Multitasking and multiprocessing programs may depend on instruction atomicity and memory operation characteristics similar to those of the VAX architecture. VEST uses special code sequences to produce exact VAX memory characteristics. VEST and the TIE cooperate to ensure VAX instruction atomicity when instructed to

do so. This mechanism is described in detail in the section Special Considerations for Instruction Atomicity.

5 Untranslatable Images

Some characteristics make OpenVMS VAX images untranslatable, including:

- Exception handler issues. Images that depend on examining the VAX processor status longword (PSL) during exception handling must be modified, because the VAX PSL is not available within exception handlers.

- Direct reference to undocumented system services. Some software contains references to unsupported and undocumented system services, such as an internal-to-VMS service, which parses image symbol tables. VEST highlights these references.

- Exact VAX memory management requirements. Images that depend on exact VAX memory management behavior do not function properly and must be modified. These images include those that depend on VAX page size or that expect certain objects to be mapped to particular addresses.

- Image format. Programs that use images as data are not able to read OpenVMS AXP images without modifications, because the image formats are different.

6 TIE Design Overview

The run-time translated image environment TIE assists in executing translated OpenVMS VAX images under the OpenVMS AXP operating system. Figure 4 and Table 1 show the contents of the TIE.

Table 1: TIE Contents

VAX-to-Alpha AXP Address Mapping (VAX State Manager)	Used to find computed destinations and other cases where VEST did not find the original VAX code. Each translated image has a mapping table included.
VAX Instruction Atomicity Controller (VAX State Manager)	Achieves VAX instruction atomicity for asynchronous events. This allows data sharing between the single asynchronous execution context (AST) provided by OpenVMS and non-AST level routines.
VAX Instruction Interpreter	Executes VAX instructions not found by VEST.
VAX Complex Instructions	Some VAX instructions do not have code generated in-line by VEST. Those instructions are processed in the TIE. Examples are MOVC3 and MOVC5 that move byte strings.
OpenVMS VAX Exception Processing	Certain aspects of OpenVMS AXP exception processing are necessarily different from OpenVMS VAX. For example, the VAX computers have two scratch registers, but Alpha AXP computers have 15. Translated condition handlers are passed the VAX equivalents.
Routines for Differences between OpenVMS VAX and OpenVMS AXP System Services	Some operating system interfaces were rearchitected. The TIE intervenes to make the differences transparent.

Problems Solved at Run Time

Complications may occur when translated OpenVMS VAX images are run under the OpenVMS AXP operating system. This section discusses the following related topics: the failure to find all code during translation, VAX instruction guarantees, instruction atomicity, memory update, and preserving VAX exceptions.

Failure to Find All Code during Translation. When the VEST binary translator encounters a branch or subroutine call to an unknown destination, VEST generates code to call one of the TIE lookup routines. The lookup routines map a VAX instruction address to a translated Alpha AXP code address. If an address mapping exists, then a transfer to the translated code is performed. Otherwise, the VAX interpreter executes the destination code. When the VAX interpreter encounters a flow of control change, it checks for returns to translated code. If the target of the flow change is translated code, the interpreter exits to this code. Otherwise, the interpreter continues to interpret the target.

Lookup operations that transfer control to the interpreter also record the starting VAX code address in an HIF file. The VAX image can then be retranslated with the HIF information, resulting in an image that runs faster.

Lookup routines are also used to call native Alpha AXP (nontranslated) routines. The TIE supplies the required special autojacketing processing that allows interoperation between translated and native routines with no manual intervention. At load time, each translated image identifies itself to the TIE and supplies a mapping table used by the lookup routines. The TIE maintains a cache of translations to speed up the actual lookup processing.

Every translated image contains both the original VAX code and the corresponding Alpha AXP code. When a translated image identifies itself, the TIE marks its original VAX addresses with the page protection called fault on execute (FOE). An Alpha AXP processor that attempts to execute an instruction on one of these pages generates an access violation fault. This fault is processed by a TIE condition handler to convert the FOE page protection into an appropriate destination address lookup operation. For example, the FOE might occur when a translated routine returns to its caller. If the caller was interpreted, then its return address is a VAX code address instead of a translated VAX (Alpha AXP code) address. The Alpha AXP processor attempts to execute the VAX code and generates a FOE condition. The TIE condition handler converts this into a JMP lookup operation.

VAX Instruction Guarantees. Instruction guarantees are characteristics of

a computer architecture that are inherent to instructions executed on that architecture. For example, on a VAX computer, if instruction 1 writes data to memory and then instruction 2 writes data to memory, a second processor must not see the write from instruction 2 before the write from instruction 1. This property is called strict read-write ordering.

The VEST/TIE pair can provide the illusion that a single CISC instruction is executed in its entirety, even though the underlying translation is a series of RISC instructions. VEST/TIE can also provide the illusion of two processors updating adjacent memory bytes without interference, even though the underlying RISC instructions manipulate four or eight bytes at a time. Finally, VEST/TIE can provide exact memory read-write ordering and arithmetic exceptions, e.g., overflow. All these provisions are optional and require extra execution time.

Tables 2 and 3 show the visibility differences between various guarantees on VAX and Alpha AXP systems as well as for translated VAX programs.

Special Considerations for Instruction Atomicity. The VAX architecture requires that interrupted instructions complete or appear never to have started. Since translation is a process of converting one VAX instruction to potentially many Alpha AXP instructions, run-time processing must achieve this guarantee of instruction atomicity. Hence, a VAX instruction atomicity controller (IAC) was created to manipulate Alpha AXP state to an equivalent VAX state. When a translated asynchronous event processing routine is called, the IAC is invoked. The IAC examines the Alpha AXP instruction stream and either backs up the interrupted program counter to restart at the equivalent VAX instruction boundary or executes the remaining instructions to the next boundary. Many VAX programs do not require this guarantee to operate correctly, so VEST emits code that is VAX instruction atomic only if the qualifier /PRESERVE=INSTRUCTION_ATOMICITY is specified when translating an image.

Table_2:_Single_Processor_Guarantees

Single Processor Guarantees Characterized by What an Observer Sees on the Same Processor That Executes the Data Change

Topic	VAX	Translated VAX	Native Alpha AXP
Instruction Atomicity	An entire VAX instruction	An entire translated VAX instruction with /PRESERVE=INSTRUCTION_ATOMICITY and TIE's instruction atomicity controller, else a single Alpha_AXP_instruction	A single Alpha AXP instruction

Table_3:_Multiple_Processor_Guarantees

Multiple Processor Guarantees Characterized by What an Observer on a Different Processor Sees versus the One Executing the Data Change

Topic	VAX	Translated VAX	Native Alpha AXP
Byte Granularity	Yes, hardware ensures this	Yes, with /PRESERVE=MEMORY_ATOMICITY	Yes, via LDx_L, merge, STx_C sequence
Interlocked Update	Yes, for aligned datum using interlock instructions	Yes, for aligned datum using VAX interlock instructions	Yes, via LDx_L, modify, STx_C

			sequence
Word Tearing	Aligned longword writes change all bytes at once Other writes are allowed to change one byte at a time	Aligned longword or quadword writes change all bytes at once	Aligned longword or quadword writes change all bytes at once

VEST-generated code consists of four sections that are detected by the IAC. These sections have the following functions:

- Get operands to temporary registers
- Operate on these temporary registers
- Atomically update VAX results that could generate side effects (i.e., an exception or interlocked access)
- Perform any updates that cannot generate side effects (e.g., register updates)

The VAX interpreter achieves VAX instruction atomicity by using the atomic move, register to memory (AMOVRM) instruction. The AMOVRM instruction is implemented in privileged architecture library (PAL) subroutines and updates a contiguous region of memory containing VAX state without being interrupted. At the beginning of each interpreted VAX instruction, a read and set flag (RS) instruction sets a flag that is cleared when an interrupt occurs on the processor. AMOVRM tests the flag, and if set, performs the update and returns a success indication. If the flag is clear, the AMOVRM instruction indicates failure, and the interpreter reprocesses the interrupted instruction.

Issues with Changing Memory. VAX instruction atomicity ensures that an arithmetic instruction does not have any partially updated memory locations, as viewed from the processor on which that instruction is executed. In a multiprocessing environment, inspection from another processor could result in a perception of partial results.

Since an Alpha AXP processor accesses memory only in aligned longwords or quadwords, it is therefore not byte granular. To achieve byte granularity, VEST generates a load-locked/store-conditional code sequence, which ensures that a memory location is updated as if it were byte granular. This sequence is also used to ensure interlocked access to shared memory. Longword-size updates to aligned locations are performed using normal load /store instructions to ensure longword granularity.

Many multiprocessing VAX programs depend on byte granularity for memory update. VEST generates byte-granular code if the condition /PRESERVE=MEMORY_ATOMICITY is specified
when translating an image. In addition, VEST generates strict read-write ordering code if the qualifier /PRESERVE=READ_WRITE_ORDERING is specified when translating an image.

Preserving VAX Exceptions. Alpha AXP instructions do not have the same exception characteristics as VAX instructions. For instance, an arithmetic

fault is imprecise, i.e., not synchronous with the instruction that caused it. The Alpha AXP hardware generates an arithmetic fault that gets mapped into an OpenVMS AXP high-performance arithmetic (HPARITH) exception. To retain compatibility with VAX condition handlers, the TIE maps HPARITH into a corresponding VAX exception when calling a translated condition handler. Most VAX languages do not require precise exceptions. For those that do,

like BASIC, VEST generates the necessary trap barrier (TRAPB) instructions if /PRESERVE=FLOATING_EXCEPTIONS is specified when translating an image.

OpenVMS AXP and OpenVMS VAX Differences

Functional Differences. Most OpenVMS AXP system services are identical to their OpenVMS VAX counterparts. Services that depend on a VAX-specific mechanism are changed for the Alpha AXP architecture. The TIE intervenes in such system services to ensure the translated code sees the old interface.

For example, the declare change mode handler ($DCLCMH) system service establishes a handler for VAX change mode to user (CHMU) instructions. The handler is invoked as if it were an interrupt service routine required to use the VAX return from interrupt or exception (REI) instruction to return to the invoker's context. On OpenVMS AXP systems, the handler is called as a normal procedure. To ensure compatibility, the TIE inserts its own handler when calling OpenVMS AXP $DCLCMH. When a CHMU is invoked on Alpha AXP computers, the TIE handler calls the handler of the translated image, using the same VAX-specific mechanisms that the handler expects.

Exception Handling. OpenVMS AXP exception processing is almost identical to that performed in the OpenVMS VAX system. The major difference is that the VAX mechanism array needs to hold the value of only two temporary registers, R0 and R1, whereas the Alpha AXP mechanism array needs to hold the value of 15 temporary registers, R0, R1, and R16 through R28.

Complex Instructions. Translating some VAX instructions would require many Alpha AXP instructions. Instead, VEST generates code that calls a TIE subroutine. Subroutines are implemented in two ways: (1) hand-written native emulation routines, e.g., MOVC5, and (2) VEST-translated VAX emulation routines, e.g., POLYH.

Together, VEST and TIE can translate and run most existing user-mode VAX binary images. As shown in Table 4, performance of translated VAX programs slightly exceeds the original goal. Performance depends heavily on the frequency of use of VAX features that are not present in Alpha AXP machines.

7 ULTRIX MIPS Translation

mx is the translator that converts ULTRIX MIPS programs to DEC OSF/1 AXP programs. The mx project started after VEST was functional, and we took advantage of the VEST common code base for much of the analysis and Alpha AXP code assembly phases of the translator. In fact, about half of the code in mx is compiled from the same source files as those used for VEST, with some architectural specifics supplied by differing include files. The code-sharing aspects of C++ have proven quite valuable in this regard.

mxr is the run-time support system for translated programs. It provides services similar to TIE, emulating the ULTRIX MIPS environment on a DEC OSF/1 AXP system. mxr is written in C++, C, and Alpha assembler.

Challenges

Creating a translator for the MIPS R2000/R3000 architecture presented us with a host of new opportunities, along with some significant challenges. The basic structure of the mx translator is much simpler than that of VEST. Both the source and the target architectures are RISC machines; therefore, the two instruction sets have a considerable similarity. Many instructions translate one for one. The MIPS architecture has very few instruction side effects or subtle architectural details, although those that are present are particularly tricky. Furthermore, the format of an executable program under the ULTRIX system collects all code in a single contiguous segment and makes it easy for mx to reliably find close to 100 percent of the code in the MIPS application. The system interfaces to the ULTRIX and DEC OSF/1 systems are similar enough that most ULTRIX system calls have functionally identical counterparts under the DEC OSF/1 system.

The challenges in mx stem from the fact that the source architecture is a RISC machine. For example, DEC OSF/1 AXP is a 64-bit computing environment, i.e., all pointers used to communicate with the operating system are 64 bits wide. This environment does not present a problem when the pointer is passed in a register. However, when a pointer (or a long data item, such as a file size) is passed in memory, it must be converted between the 32-bit representation, used by the ULTRIX system, and the 64-bit AXP representation, even when the semantics of the operating system call are the same on both systems.

A significant challenge is the fact that our users' expectations for performance of translated programs are much higher than for VEST. Reasoning that the source and target machines are similar, users also expect mx to achieve a translated program performance better than that of the source program, since Alpha AXP processors are faster. Thus, as our performance goal, we set out to produce a translated program that runs at about the same speed as the original program would run on an MIPS R4000 machine with a 100-megahertz (MHz) internal clock rate.

Mapping the Architectures

At first glance, it appears that we could simply assign each MIPS register to a corresponding Alpha AXP register, because each machine has 32 general-purpose registers. The translated code would then have two scratch registers, since the MIPS architecture does not allow user-level programs to use registers K0 and K1, which are reserved for the operating system kernel.

Unfortunately, translation requires more than two scratch registers. The Alpha AXP architecture does not have byte or halfword (16-bit) loads or stores, and the code sequences for performing these operations require

four or five scratch registers. Furthermore, mx requires a base register to locate mxr without having to load a 64-bit address constant at each call. Finally, the MIPS architecture has more than 32 registers, including the HI and LO registers used by the multiply and divide instructions,

and a floating-point condition register, whose layout and contents do not correspond to the Alpha AXP floating-point condition register.

In mx, we assign registers using standard compiler techniques. To assign registers to 33 MIPS resources (the 32 general registers plus one 64-bit register to hold both HI and LO), certain registers are permanently mapped, and other MIPS registers are kept in either AXP registers or memory. The MIPS argument-passing registers A0 through A3 are permanently assigned to Alpha AXP registers R16 through R19, which are the argument registers in the DEC OSF/1 AXP calling standard. This correspondence simplifies the work needed when mxr must take arguments for an ULTRIX system call and pass them to a DEC OSF/1 system call. Similarly, the argument return registers V0 and V1 are mapped to the Alpha AXP argument return registers R0 and R1. The return address registers and stack pointer registers of the two machines are also mapped. MIPS R0 is mapped to Alpha AXP R31, where both registers contain the same hard-wired zero value. We reserve Alpha AXP registers R22 through R24 as scratch registers and also use them when interfacing to mxr. We reserve Alpha AXP R14 as a pointer to an mxr communication area. Finally, we reserve three more registers as scratch registers for use by the code generator.

The remaining 16 Alpha AXP registers are available to be assigned to the remaining 23 MIPS resources. After the code is analyzed and we have register usage information, the 16 most frequently used MIPS registers get mapped to the remaining 16 Alpha AXP registers, and the remaining registers are assigned to memory slots in the mxr communication area. When a MIPS basic block uses one of the slotted registers, mx assigns it to one of the scratch registers. If the first reference reads the old contents of the register, mx generates a load instruction from the communications area. If the value of the MIPS resource changes in the basic block, the scratch register is stored in the communication area before the end of the block. As in most compilers, if we run out of registers, a spill algorithm chooses a value to save in the communication area and frees up a register.

Alpha AXP integer registers are 64 bits wide, whereas MIPS registers are only 32 bits wide. We chose to keep all 32-bit values in Alpha AXP integer registers as sign-extended values, with the high 32 bits equal to bit 31. This approach occasionally requires mx to generate additional code to create canonical 32-bit integer results, but the 64-bit compare operations do not need to change the values that they are comparing.

The floating-point architecture is more complex. Each of the 32 MIPS floating-point registers is 32 bits wide. Only the even registers are used for single precision, and a double-precision number is kept in an even-odd register pair. We map each pair of MIPS floating-point registers onto a single 64-bit Alpha AXP floating-point register. Also, one Alpha AXP floating-point register represents the condition code bit of the MIPS

floating-point control register. Thus, the mx code generator can use 14 scratch registers. mx goes to considerable effort to find paired loads and stores in the MIPS code stream, and to merge them into one Alpha AXP floating-point operation.

MIPS single-precision operations cause problems with floating-point correspondence. Since on MIPS machines, the single-precision number is kept in only the even register of the register pair, the even and odd registers in a pair are independent when single-precision (or integer) operations are done in the floating-point unit. On Alpha AXP machines, computation must be done on a value extended to double format in the whole 64-bit register. We defined two forms for values in Alpha AXP floating-point registers: computational form, in which computation is done, and canonical form, which mimics the MIPS even and odd registers. If a MIPS program loads an even register and uses this register as a single-precision value, mx loads the value from memory to be used computationally. If a MIPS program loads only an even register but does not use this register in the basic block, mx puts the 32-bit value into half of the Alpha AXP floating-point register. This permits correct behavior in the pathological case where half of a floating-point number is loaded in one place, and the other half is loaded in some other basic block. If a register is used as a single-precision number in a basic block without first being loaded, the code generator inserts code to convert it from canonical to computational floating-point form. If a single-precision value has been computed in a block and is live at the end of the block, it is converted to canonical form.

mx inserts a register mapping table into the translated program that indicates which MIPS resources are statically mapped to which Alpha AXP registers, and which MIPS resources are normally kept in memory. This table allows mxr to find the MIPS resources at run time.

Finding Code

As with the VEST translator, mx finds code by starting at entry points and recursively tracing down the flow of control. mx finds entry points using the executable file header, the symbol table (if present), and feedback from mxr (if present). Finally, mx performs a linear scan of the entire text section for unexamined words. mx analyzes any data that looks like plausible code but does not connect this data into the main flow graph. Plausible code consists of a series of valid MIPS instructions terminated by an unconditional transfer of control.

While finding code and connecting the basic blocks into a flow graph, mx looks for the code sequence that indicates a switch statement, i.e., a multi-way branch, usually through an element of a table. mx finds the branch table and connects each of the possible targets as successors of the branch.

Code Analysis

Our static analysis of hundreds of MIPS programs indicates that only 10 instructions account for about 85 percent of all code. These

instructions are LW, ADDIU, SW, NOP, ADDU, BEQ, JAL, BNE, LUI, and SLL. The corresponding sequences of Alpha AXP code range from zero operation codes, or opcodes, (for NOP, since the Alpha AXP architecture does not require NOPs anywhere in the code stream) to two opcodes (for SLL).

Binary Translation

Code analysis for source programs is much more important in mx than in VEST, because the coding idioms for many common operations differ between the Alpha AXP and MIPS processors. The simple technique of mapping each MIPS instruction to a sequence of one or more Alpha AXP instructions loses much of the context information in the original program.

For example, the idiom used to load a 32-bit constant into a register on MIPS machines is to generate a load upper immediate (LUI) opcode, placing a 16-bit constant in the high-order 16 bits of a register. This operation is followed by an OR immediate (ORI) opcode, logically ORing a 16-bit zero-extended value into the register. The LUI corresponds exactly to the Alpha AXP load address high (LDAH) opcode. However, the Alpha AXP architecture has no way of directly ORing a 16-bit value into a register and cannot even load a zero-extended 16-bit constant into a register. When the high-order bit of the 16-bit constant is 1, the shortest translation for the ORI is three instructions. The mx translator scans the code looking for such idioms, and generates the optimal two-instruction sequence of Alpha AXP code that performs the 32-bit load. No opcode exists that corresponds to the ORI, but the results in the registers are correct.

When we started writing the mx translator, we listed a number of code possibilities that we thought we would never see. In retrospect, this was a misguided assumption. For example, we have seen programs that branch into the delay slot of other instructions, requiring us to indicate that the delay slot instruction is a member of two different basic blocks - the block it ends, and the one it starts. We have observed programs that put software breakpoint (BREAK) instructions in the branch delay slot, and thus BREAK ends a basic block without being the last instruction. Some compilers schedule code so that half of a floating-point register is stored and then reused before the other half is stored. The general principle that we intuit from these observations is "if a code sequence is not expressly prohibited by the architecture, some program somewhere will use it."

Code Generation

After the program is parsed and analyzed and the flow graph is built, the code generator is called. It builds the register mapping table and then, in turn, processes each basic block, generating Alpha AXP code that performs the same functions as the MIPS code.

At each subroutine entry, mx scans the code stream with a pattern-matching algorithm to see if the code corresponds to any of a number of standard MIPS library routines, such as strcpy. (Note that the ULTRIX operating system has no shared libraries, so library routines are bound into each binary image.) If a correspondence exists, the entire subroutine is recursively deleted from the flow graph and replaced with a canned routine to perform the subroutine's work on Alpha AXP processors. This technique

contributes significantly to the performance of translated programs.

For each remaining basic block, the instructions are converted to a linked list of intermediate opcodes. At first, each opcode corresponds exactly to a MIPS opcode. The list is then scanned by an optimization phase, which looks for MIPS coding idioms and replaces them with abstract machine instructions that better reflect the idiom. For example, mx changes loads of immediate values to a non-MIPS hardware load immediate (LI) instruction; shift and add sequences to abstract operations that reflect the Alpha AXP scaled add and subtract sequences; and sequences that change the floating-point rounding mode (used to truncate a floating-point number to an integer) to a single opcode that represents the Alpha AXP convert operation with the chopped mode (/C) modifier.

MIPS code contains a number of common code sequences that cross basic block boundaries, but which can be compressed into a single basic block in Alpha AXP code. Examples of these are the min and max functions, which map neatly onto a single conditional move (CMOVxx) instruction in Alpha AXP code. The code generator looks for these sequences, merges the basic blocks, and creates an extended basic block, which includes pseudo-opcodes that indicate the MIPS code idiom.

After the optimizer completes the list of instructions, it translates each abstract opcode to zero or more Alpha AXP opcodes, again building a linked list of instructions. This process may permit further improvements, so the optimizer makes a second pass over the Alpha AXP code.

When processing a basic block, the code generator assumes that it has an unlimited number of temporary resources. Since this is not actually true, the code generator then calls a register assigner to allocate the real Alpha AXP temporary resources to the intermediate temporary registers. The register assigner will load and spill MIPS resources and generated temporary registers as needed.

Finally, the list of Alpha AXP instructions is assembled into a binary stream, and the instruction scheduler rearranges them to remove resource latencies and use the chip's multiple issue capability.

Image Formats

The file format for input is the standard ULTRIX extended common object file format (COFF). In most ULTRIX MIPS programs, the text section starts at 00400000 (hexadecimal) and the data at 10000000 (hexadecimal). In virtually all programs, a large gap exists between the virtual address for the end of text and the start of the data section. When mx creates the output image, it places the generated Alpha AXP code after the MIPS code and before the MIPS data. This allows the program to have one large text section. The Alpha AXP code begins at an Alpha AXP page boundary, so that we can set the memory protection on the MIPS code separately from the Alpha

AXP code.

The translated image is not in DEC OSF/1 AXP executable format. Instead, it looks like a MIPS COFF file, but with the first few bytes changed to the string "#!/usr/bin/mxr".

Executing a Translated Program

When a translated image is run on DEC OSF/1 AXP, its modified header invokes mxr first. mxr uses the memory map (mmap) system call to load the translated program at the same virtual address that it would have had under the ULTRIX operating system. mxr resets the protection of the MIPS code to read/no-write/no-execute, the Alpha AXP code to read/no-write/execute, and the data to read/write/no-execute.

mxr allocates a communication area and initializes Alpha AXP R14 to point to this area. The communication area contains save areas for MIPS resources, initialized pointers to mxr service routines, and other scratch space. mxr then constructs new command argument (argv) and environment vectors as 32-bit wide pointers (as the MIPS program expects), arranges to intercept certain signals from the DEC OSF/1 AXP system, and transfers control to the translated start address of the program.

When a system signal is delivered to the program, control goes to the signal intercept code in mxr. This code transforms the signal context structure from the DEC OSF/1 AXP system and constructs an ULTRIX MIPS style context, which it then passes to the translated signal handler.

Certain signals are processed specially. For instance, a program that attempts to transfer control to a location containing MIPS code rather than translated code gets a segmentation violation, since the MIPS code is not executable. This situation can occur if a routine modifies its return address to be a MIPS address constant. mxr will examine the target address and, if it corresponds to the start of a pretranslated MIPS basic block, divert the flow of control to the translated code for that block. If not, mxr enters the MIPS interpreter. The interpreter proceeds to emulate the MIPS code until a translated point is reached. mxr then resynchronizes its machine state and reenters the translated code.

Translation Goals and Classes of Programs Not Supported

Our goal was to translate most user mode MIPS programs compiled for a MIPS R2000 or R3000 machine running ULTRIX Release 4.0 (or later) to run identically on the DEC OSF/1 AXP system with acceptable performance. As shown in Table 5, performance of translated MIPS programs meets or exceeds the original goal.

Due to extreme technical obstacles, some classes of programs will never be supported by mx. We decided not to translate programs that use privileged opcodes or system calls or that need to run with superuser privileges. In cases where the file system hierarchy differs between the ULTRIX and DEC OSF/1 AXP systems, programs that expect files to be in particular places or in a particular format may fail. Similarly, programs that read /dev/kmem and expect to see a ULTRIX MIPS memory layout fail.

Certain other classes of programs are not currently supported, but are technically feasible. These include big endian MIPS programs from non-Digital MIPS environments, programs that use R4000 or R6000 instructions that are not present on the R3000 model, programs that need to be multiprocessor safe, and programs that require certain categories of precise exception behavior.

8 Summary

Building successful turnkey binary translators requires hard work but not magic. We built two different translators, VEST and mx. In both cases, the old and new environments are, by design, quite similar in fundamental data types, memory addressing, register and stack usage, and operating system services. Translators between dissimilar architectures or operating systems are a different matter. Translating the code might be a reasonably straightforward task. However, emulating a run-time environment in which to execute the code might present insurmountable technical and business obstacles. Without capturing the environment, an instruction translator would be of no use.

The idea of binary translation is becoming more common in the computer industry, as various other companies start on their transitions to 64-bit architectures.

9 Acknowledgments

Steve Hobbs originally suggested the binary translation path in the architecture task force discussions. Nancy Kronenberg and Bob Supnik added critical early support and later coordination. Jud Leonard set the engineering direction of doing careful static translation once, instead of on-the-fly dynamic translation at each execution. Butler Lampson boosted morale at a critical time. Jim Gettys has also been an important and vocal supporter.

The success of the translators would not have been possible without the enthusiastic support of the OpenVMS AXP and DEC OSF/1 AXP operating system groups, and the respective run-time library groups, especially Matt LaPine, Larry Woodman, Hai Huang, Dan Murphy, Nitin Karkhanis, Ray Lanza, Anton Verhulst, and Terry Grieb.

The Porting and Performance Engineering Group did extensive porting and testing of customer applications. The group members, especially Shamin Bhindarwala and Robi Al-Jaar, were sources of extremely valuable customer feedback. The Engineering System Group under Mike Greenfield also made extensive early use of the translators and provided valuable feedback.

The Alpha AXP Migration Tools team is relatively small for the substantial amount of work accomplished in the past two and one-half years. Every person has made several key contributions. In addition to the authors of this paper, the team members are: Kate Burleson, Peigi Cleminshaw, George Darcy, Catherine Frean, Bruce Gordon, Rick Gorton, Kevin Koch, Mark Herdeg, Giovanni Della Libera, Nikki Mirghafori, Srinivasan Murari, Jim Paradis, and Ashutosh Roy.

10 References and Note

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

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

3. C. Hunter and J. Banning, "DOS at RISC," Byte Magazine (November 1989): 361-368.

4. Echo Logic, News Release (May 4, 1992).

5. L. Wirbel, "DOS-to-UNIX Compiler," Electronic Engineering Times (March 14, 1988): 83.

6. A. Bergh, K. Keilman, D. Magenheimer, and J. Miller, "HP 3000 Emulation on HP Precision Architecture Computers," Hewlett-Packard Journal (December 1987).

7. Datum is the term used to refer to a piece of information that has an address and a size.

   Alignment is the property of a datum of size 2[n] bytes. This datum is aligned if its byte address has n low-order zeros. A size or address not meeting this constraint implies that the datum is unaligned.

   Instruction atomicity is the property of instruction execution on single processor systems such that an interrupted instruction has been completed or has never started, i.e., partial execution of an instruction is never observed.

   Granularity is the property of memory writes on multiprocessor systems

such that independent writes to adjacent aligned data produce consistent results. The terms byte, word, longword, quadword, and octaword granularity refer to writing 1-, 2-, 4-, 8-, and 16-byte size adjacent data.

Interlocked update is the property of memory updates (read-modify-write sequences) on multiprocessor systems such that simultaneous independent updates to the same aligned datum will be consistent. This property causes serialization of the independent read-modify-write sequences and is not guaranteed for an unaligned datum.

Word tearing is the property of aligned memory writes on multiprocessor systems such that a reader independent of the writer can see partial results of the write.

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

11 Trademarks

The following are trademarks of Digital Equipment Corporation:

ALL-IN-1, Alpha AXP, AXP, DEC, DEC 3000 AXP, DEC 7000 AXP, DEC OSF/1 AXP, DECstation, Digital, OpenVMS AXP, OpenVMS VAX, PDP-11, ULTRIX, and VAX.

The following are third-party trademarks:

HP is a registered trademark of Hewlett-Packard Company.

Macintosh is a registered trademark of Apple Computer, Inc.

MIPS is a trademark of MIPS Computer Systems, Inc.

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

Windows is a trademark of Microsoft Corporation.

12 Biographies

Richard L. Sites Dick Sites is a senior consultant engineer in the Semiconductor Engineering Group, where he is working on binary translators and the Alpha AXP architecture. He joined Digital in 1980 and has contributed to various VAX implementations. Previously, he was employed by IBM, Hewlett-Packard, and Burroughs, and taught at the University of California. Dick received a B.S. in mathematics from MIT and a Ph.D. in computer science from Stanford University. He also studied computer architecture at the University of North Carolina. He holds a number of patents on computer hardware and software.

Anton Chernoff Anton Chernoff is a member of the technical staff at Digital Equipment Corporation, working in the Alpha AXP Migration Tools Group. He joined Digital in 1991, but also worked at Digital between 1973 and 1981

as project leader and developer of the RT-11 and RSTS/E operating systems. Anton spent 1982 through 1991 at Liant Software Corporation as a senior consulting engineer in compiler and debugger development.

Matthew B. Kirk Matthew Kirk is a senior software engineer in the SEG/AD AXP Migration Tools Group, where he works on binary translator development, testing, and support. He joined Digital in 1986 and has also designed and developed automated architectural test software for pipelined VAX hardware and the CI computer interconnect. Matthew holds a B.S. in computer science (1986) from the University of Massachusetts.

Maurice P. Marks Maurice Marks is a senior engineering manager in the Semiconductor Engineering Advanced Development Group. He currently manages the AXP Migration Tools Group and contributed to the design and implementation of the translators. In Maurice's twenty years with Digital, he has led compiler, operating system, hardware and software tools, CAD, system, and chip projects. He holds B.Sc. and B.E. degrees from the University of New South Wales and has published papers on transaction processing, software portability, and CAD technology. Maurice is a member of the Australian Computer Society.

Scott G. Robinson Scott Robinson is a software engineering manager in the AXP Migration Tools Group. He contributed to the design and implementation of the binary translators, particularly the VAX translated image environment. Scott has also developed implementations of DECnet and CAD/CAM systems to design VAX processors. Prior to joining Digital in 1978, Scott worked on a variety of Digital hardware and software implementations. He holds a B.S. in electrical engineering from the University of Arizona and is a member of IEEE.