DIGITAL FX!32: Combining Emulation and Binary Translation

Raymond J. Hookway
Mark A. Herdeg

The DIGITAL FX!32 software product uniquely combines emulation and binary translation to enable any 32-bit application that executes on an Intel x86 microprocessor running the Windows NT 4.0 operating system to be installed and to execute on an Alpha microprocessor running Windows NT 4.0. Benchmark tests indicate that after translation, x86 applications run as fast on a 500-MHz Alpha system with DIGITAL FX!32 software installed as on a 200-MHz Pentium Pro system. The emulator and its associated runtime software provide transparent execution of applications written for x86-based platforms. The emulator produces profile data that is used by the translator and takes advantage of translation results as they become available. The translator provides native Alpha code for the portions of an x86 application that have previously been executed. A server manages the translation process for the user, making the process completely transparent.

Three factors contribute to the success of a microprocessor: price, performance, and software availability. The DIGITAL FX!32 product addresses the third factor, software availability, by making hundreds of new applications available on Alpha-based platforms running the Windows NT operating system. DIGITAL FX!32 software combines emulation and binary translation to provide fast, transparent execution of Intel x86 applications on Alpha systems.

Since its introduction in 1992, the Alpha microprocessor has been the fastest microprocessor available. A large number of native applications are available on Alpha systems, particularly those applications that require a high-performance processor. With the introduction of DIGITAL FX!32 software, 32-bit programs that can be installed and executed on x86 systems running the Windows NT 4.0 operating system can also be installed and executed on Alpha systems running Window NT 4.0. Except for having to specify that a program is an x86 application, installing and running an application is the same on an Alpha system as on an x86 system. The performance of an x86 application running on a high-end Alpha system is similar to the performance of the same application running on a high-end x86 system.

A number of systems have successfully used emulators to run applications on platforms for which the applications were not initially targeted.[1,2] The major drawback has been poor performance.[2] Several emulators have used dynamic translation, translating small segments of a program as it is executed, to achieve better performance than that obtained by an interpreter alone.[2–4] Dynamic translation involves a basic trade-off between the amount of time spent translating and the resulting benefit of the translation. If an emulator spends too much time on the translation and related processing, the executing program will be unresponsive. This limits the optimizations that can be performed by the emulator using dynamic translation.

FX!32 overcomes the performance problem by not doing any translation while the application is executing. Rather, FX!32 captures an execution profile that is later used by a binary translator[5] to translate into native Alpha code those parts of the application that have been executed. Since the translator runs in the back-

ground, it can use computationally intensive algorithms to improve the quality of the generated code. To our knowledge, FX!32 is the first system to explore this combination of emulation and binary translation.

In this paper, we describe how FX!32 works. We begin with an overview and discuss each of the major components in more detail. We then present some benchmark test results and briefly describe several limitations of the current version of DIGITAL FX!32 software.

Overview

On Alpha systems, the Windows NT operating system uses an emulator to run 16-bit x86 applications. These applications can be installed and run in the same way as they are installed and run on x86 systems, but the execution is slower. The emulator built into FX!32 provides a similar capability for 32-bit x86 applications.

Unlike the emulation software in the 16-bit environment, FX!32 provides a binary translator that translates 32-bit x86 applications into native Alpha code. The translation is done in the background and requires no user interaction. Using background translation allows the translator to perform optimizations that, in terms of computational resources, would be too expensive to accomplish while an application is running. An application translated by means of FX!32 runs up to 10 times faster than the same application running under the emulator.

DIGITAL FX!32 software consists of the following seven major components:

1. The transparency agent, which provides for transparent launching of 32-bit x86 applications.
2. The runtime, which loads x86 images and sets up the run-time environment to execute them. As part of loading an image, the runtime component jackets imported application programming interface (API) routines. Jackets are small code fragments that allow the x86 code to call Alpha Windows NT API routines.
3. The emulator, which runs an x86 application making use of translated code when it is available.
4. The translator, which produces a translated image using profile information received from the emulator.
5. The database, which stores execution profiles produced by the emulator and used by the translator. Translated images are also stored in the database, along with configuration information.
6. The server, which maintains the database and runs the translator as appropriate.
7. The manager, which allows the user to control resources used by the DIGITAL FX!32 software.

Figure 1 shows the relationships between these major components, each of which is discussed in more detail in the sections that follow.

The Transparency Agent

The transparency agent provides for transparent launching of 32-bit x86 applications. Launching an application on the Windows NT operating system always results in a call to the CreateProcess API routine. By hooking calls to this routine, the transparency agent can examine every image as it is about to be executed. If a call to CreateProcess specifies that an x86 image is to be executed, the transparency agent invokes the runtime component to execute the image.

FX!32 inserts the transparency agent into the address space of each process. A process that contains the trans-

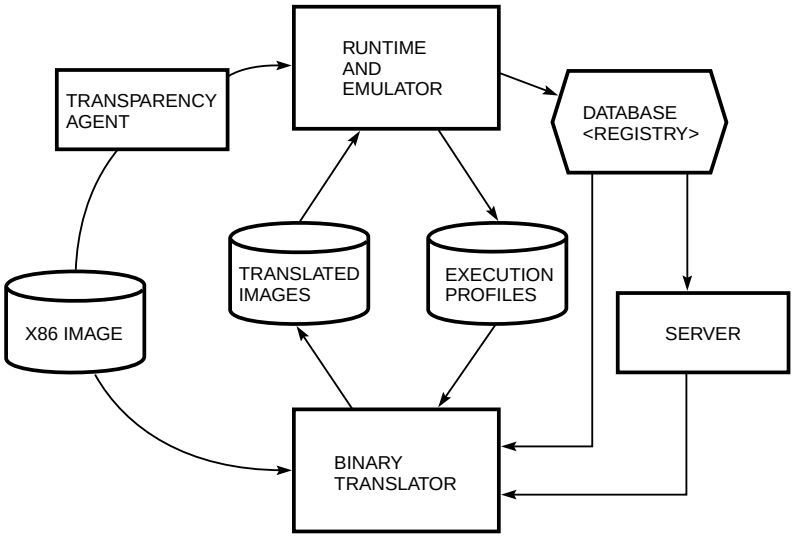

Figure 1
DIGITAL FX!32 System Components

parency agent is said to be enabled. Once a process is enabled, any attempt to execute an x86 image causes the runtime to be invoked to execute the process. The agent is propagated through the system because each attempt to create a process to run an Alpha image results in that created process being enabled.

By the time a user is logged on, FX!32 has enabled all the top-level processes, and any attempt to execute a 32-bit x86 application invokes the runtime component. The initial processes that are enabled are the Windows shell (explorer.exe), the service control manager (services.exe), and the remote procedure call server (rpcss.exe). When FX!32 is installed, the fx32strt.exe file is registered as the Windows shell. When a user logs on, fx32strt.exe runs and enables the real Windows shell, explorer.exe. The FX!32 server enables the service control manager when it starts, usually when the system is booted. Currently, any service process that is started by the service control manager before the server is started is not enabled. (The only exception is rpcss.exe, which is explicitly enabled by the server). We hope to alleviate this limitation in a future version of the DIGITAL FX!32 software.

Processes are enabled using a technique described by Jeffrey Richter in Chapter 16 of his book *Advanced Windows NT*[6] to inject a copy of the transparency agent into the process' address space.

The Runtime

The transparency agent invokes the runtime whenever an attempt is made to execute an x86 image. The runtime loads the image into memory, sets up the runtime environment required by the emulator, and then calls the emulator to execute the image.

The runtime replaces the Windows NT loader, which can only load Alpha images; the Windows NT loader returns an error reporting an image of the wrong architecture if it is invoked to load an x86 image. The runtime duplicates the functionality of the Windows NT loader, which includes relocating images that are not loaded at their preferred base address, setting up shared sections, and processing static thread local storage sections.

The runtime registers each image it processes with the Windows NT operating system by inserting pointers to that image into various lists that are used internally by the system. Maintaining these lists allows the native Windows NT code to correctly implement API routines, such as LoadResource and GetModuleHandle, which require access to images that have been loaded. The registration also ensures that the DllMain functions of the loaded dynamic link libraries (DLLs) are called as appropriate. (The entry points of x86 DLLs are jacketed by the runtime.)

Fortunately, the image lists that FX!32 must modify are in the user's address space, and no modification of the Windows NT operating system was required to register images with the system. Unfortunately, the structure of these lists is not part of the documented Win32 interface, and using them creates a dependency on the Windows NT version that is being run. FX!32 has dependencies on a number of undocumented features of the Windows NT operating system. Although the DIGITAL FX!32 product is more dependent on a particular version of the operating system than a typical layered application is, it is remarkable that the implementation of FX!32 did not require any changes to the Windows NT operating system.

The runtime also registers the image in the FX!32 database. This database maintains information about x86 images that have been loaded, including the application that loaded the image, profile data that was produced by the interpreter, and any translation of the image. The runtime accesses the database with a unique image identifier (ID), which the runtime obtains by hashing the image's header. Therefore, the image ID is determined by the content of the image, not by its location in the file system, and the information that FX!32 associates with the image can be accessed independently of the image's location on the disk. For example, if an application is installed in one directory and some of the images loaded by the application are subsequently translated by FX!32, the translated images will be located by FX!32 even if the application is later installed in a different directory.

When the runtime finds a translated image in the database, it loads this image along with the corresponding x86 image. Translated images are normal DLLs, loaded by the native LoadLibrary API routine. Translated images contain additional sections that store information required by the runtime to map x86 routines to the corresponding Alpha code.

The runtime duplicates the Windows NT loader function of binding an image's imports, using symbolic information in the image to locate the address of the imported routine or data. The runtime treats imports that refer to entries in Alpha images specially, however, by redirecting the imports to refer to the correct jacket entry in the FX!32 DLL, jacket.dll.

The jacket routines in jacket.dll enable an x86 user program to call the native Alpha implementation of the Win32 API. These jacket routines are extremely important because they allow x86 applications to use high-performance code that has been tuned to the Alpha platform. Some x86 applications run faster on the Alpha platform than on the x86 platform, even without being translated, because of the large amount of time the applications spend in native DLLs.

Each jacket contains an illegal x86 instruction that serves as a signal to the interpreter that a change is to be made to the Alpha environment. The interpreter calls an Alpha jacket routine at a fixed offset from the illegal x86 instruction. The basic operation of most

jacket routines is to move arguments from the x86 stack to the appropriate Alpha registers, as dictated by the Alpha calling standard. Some jacket routines provide special semantics for the native routine being called, as required by FX!32. For example, the jacket for the GetSystemDirectory routine returns the path to the FX!32 directory rather than the path to the true system directory so that x86 applications do not overwrite native Alpha DLLs.

For an x86 application to run under FX!32, every image it loads must be either an x86 image or an Alpha image for which jackets exist. Therefore, FX!32 provides jackets for all the DLLs that implement the Win32 interface and for many redistributable DLLs. FX!32 currently provides jackets for more than 50 native Alpha DLLs, which has enabled the FX!32 development team to run almost all the commercial applications tested. Each new release of DIGITAL FX!32 software provides additional jackets, and the developers intend to jacket new interfaces as they are released.

The Emulator

The fundamental job of the emulator is to run x86 applications before they are translated. The first time an x86 image executes under FX!32, the image is executed by the emulator.

The emulator also serves as a backup for translated code. Because it is not possible to statically determine all the code that can ever be executed by an application (especially for applications that generate code on-the-fly), the emulator is always present to execute such untranslated x86 application code. Previous binary translators built by DIGITAL also depended on the presence of an emulator in this role.[5] Emulator performance is more of an issue for FX!32 because, unlike those earlier binary translators, all application code is interpreted when the x86 application is first run.

The emulator is an Alpha assembly language program that interprets the subset of x86 instructions that can be executed by a Win32 application. While an x86 application is running, the x86 processor state is kept partially in Alpha registers and partially in a per-thread data structure called the CONTEXT. The x86 integer registers are permanently mapped to Alpha registers, and Alpha registers store the state of the x86 condition codes. While the emulator is running, a dedicated Alpha register points to the CONTEXT. The CONTEXT stores the x86 per-thread processor context and any part of the x86 processor state that must be maintained across calls to other parts of the system, for example, calls to Alpha API routines.

Pipelined Dispatch

The structure of the emulator is a classic fetch-and-evaluate loop. The emulator dispatches on the first two bytes of each instruction, performing the lookup in a table of 64K entries. Each entry contains the address of the routine to execute to interpret an instruction and the length of the instruction.

The structure of the dispatch loop has been carefully crafted to make efficient use of 64-bit Alpha registers and to efficiently schedule the execution of code in the loop. Software pipelining is used to overlap the fetch and dispatch table lookup for the next instruction with the execution of the current instruction. At the top of the loop, at least eight bytes, starting at the address of the current instruction, are in Alpha registers. Length information from the dispatch table determines the first two bytes of the next instruction, allowing the dispatch table lookup to be overlapped with the execution of the current instruction. A fetch of additional bytes from the instruction stream is also initiated. Finally, the loop dispatches to the routine whose address was obtained from the table on the previous iteration of the loop.

The individual routines have been factored by using subroutines and coroutines to perform operations like operand fetching, making them as small as possible. As a result, the emulator code required to execute the most frequently executed x86 instructions fits in the first-level cache.

Condition Code Evaluation

Condition codes are generated by the execution of many of the x86 instructions. We have observed that condition codes are frequently set and relatively infrequently examined. The emulator takes advantage of this by evaluating the condition codes only when they are used, that is, by using a "lazy evaluation" technique. The execution of a typical instruction saves only enough state to allow the evaluation of condition codes, if required, at a later time. This takes much less effort than initially evaluating the condition codes. The additional advantage in deferring the evaluation is that only the condition codes that are used need to be generated. For example, the overflow condition code may never be computed if only the zero flag is used.

Floating-point Instruction Emulation

The 80-bit x86 floating-point registers are modeled by a stack of 64-bit memory locations that contain floating-point values. The decision to use 64-bit intermediate values, rather than to faithfully replicate the 80-bit model, was based on the need to achieve good performance when executing x86 floating-point code on the Alpha processor. This decision was supported by the fact that the Windows NT operating system also uses a 64-bit floating-point model. Although this is an approximation, our experience to date has shown that this was a good compromise. Very few applications rely on the full precision provided by the x86 floating-point unit's (FPU's) 80-bit registers.

The emulator also implements a somewhat simplified model of the x86 FPU's register file. Most instructions use the x86 FPU register file as a traditional operand stack; however, several instructions can create a register file state that is not strictly a stack by freeing registers in the middle of the stack, by moving the stack pointer without pushing or popping, or by initializing the register file in a way that breaks the stack model. Modeling the full complexity of the x86 FPU register file would be extremely expensive, and experience has shown that almost all programs use the register file strictly as a stack. The current version of the emulator takes advantage of this. We are investigating ways to model the floating-point registers in a way that maintains good performance but does not depend on their being treated as a stack.

Generation of Profiles

While it is interpreting an x86 program, the emulator generates profile data for use by the translator. The profile data includes the following information:

- Addresses that are the targets of call instructions
- (*Source address, target address*) pairs for indirect control transfers
- Addresses of instructions that make unaligned references to memory

The translator uses this information to generate *routines,* that is, units of translation that approximate a source code routine. The emulator generates profile data by inserting values in a hash table whenever a relevant instruction is interpreted. For example, as part of interpreting the call instruction, the emulator makes an entry in a hash table that records the target of the call. When an image is unloaded (either as a result of a call on the FreeLibrary routine or when the application exits), the runtime processes the hash table to produce a profile file for that image. This profile is processed by the server and can result in the server invoking the translator to create a new translation of the image.

To detect available translated code, the emulator uses the same hash table that it employs to gather the profile data. The x86 addresses for which there are translated routines and the address of the corresponding translated code are entered into the hash table by the runtime when it loads an x86 image that has been translated. When a call instruction is interpreted, the emulator looks up the target address. If a corresponding translated address exists, the emulator transfers control to that address.

The Translator

The server invokes the translator to translate x86 images for which a profile exists in the database. The translator uses the profile to produce a translated image. On subsequent executions of the image, the translated code is used, substantially speeding up the application.

Structure and Order of Operations

The translator has eight major components (or phases): the regionizer, build, the register mangler, the condition code mangler, improve, the code selector, the scheduler, and the assembler. (An additional phase that performs various peephole optimizations is disabled in the DIGITAL FX!32 V1.0 translator.) The major components function as follows:

1. The Regionizer—The regionizer uses data in the profile to divide the source image code into routines, which are described in the section Generation of Profiles. Each call target in the profile is used to generate an entry to a routine. The regionizer represents routines as a collection of regions. Each region is a range of contiguous addresses, which contains instructions that can be reached from the entry address of the routine. Unlike basic blocks, regions can have multiple entry points. The smallest collection of regions that contain all the instructions that can be reached from the routine entry is used to represent the routine. Many routines have a single region. This representation was chosen to efficiently describe the division of the source image into units of translation.

   The regionizer builds routines by following the control flow of the source image. When an indirect jump instruction is encountered while following the control flow, the possible targets of the instruction are obtained from the profile. Without this profile information, it would be very difficult to reliably identify these targets, and indirect jumps would have to be treated as returns from the routine. The profile information makes it possible to reliably generate a more complete representation of routines with correct control flow.

After the regionizer runs, each of the other major components is run in sequence for each routine.

2. Build—Build reparses the x86 instructions in the routine to create an internal representation (IR) of the routine for use by the subsequent components. The IR is a graph of basic blocks and is similar to the IR used by many optimizing compilers.
3. The Register Mangler—The initial IR is a straightforward representation of the source x86 code. This representation ignores the overlap of the x86 registers; the IR treats each occurrence of EAX, AX, AH, and AL as a separate register. The register mangler adds insert and extract operations as necessary to represent the actual semantics of the x86 registers.

4. The Condition Code Mangler—The effect of x86 instructions on condition codes is represented implicitly in the initial IR. The condition code mangler adds instructions to explicitly generate condition codes. Since the condition code mangler understands the control flow of the entire routine, it knows when condition codes are live and only adds code to generate condition codes when they are used later in the routine.
5. Improve—Improve performs several transformations that produce code more suited to the Alpha architecture. In the initial IR, each push and pop instruction is explicitly represented as a decrement/increment of the x86 stack pointer, accompanied by a store/load. Improve collects all the manipulation of the x86 stack pointer into a single decrement at the beginning of a basic block and a single increment at the end of that block. Improve also uses simple value numbering and analysis of memory references to try to eliminate loads and stores to both the x86 stack and the floating-point stack and to perform constant folding. Although Improve performs only relatively simple optimizations on a single basic block, we have found it to be quite effective in improving the quality of the code that is generated.
6. The Code Selector—The code selector transforms the IR from a representation that contains mostly x86 instructions to one that contains only Alpha instructions. This transformation is done instruction by instruction, with each x86 instruction being replaced by a sequence of Alpha instructions that produce the same effect. The implementation of the code selector is based on the TWIG code generator.[7] Although the code selector is capable of dealing with much more complicated patterns of instructions, this capability is not currently used.
7. The Scheduler—After the code selector is run, all the instructions in the IR are Alpha instructions. The scheduler reorders the instructions within a basic block to minimize the cycle count for the target processor.
8. The Assembler—The assembler builds the output translated image.

Use of Profile Data

The regionizer is the only component of the current translator that uses the control flow information in the profile. The regionizer uses the profile to determine which parts of the source image are translated. Future versions of the translater will use the profile to perform path-directed optimizations and to place code so as to reduce cache misses. Those changes will improve the performance of translated code.

Retranslation of an image is triggered by growth in the size of the profile. Because profile data is generated only when the emulator executes previously untranslated parts of the source image, an increase in the size of the profile indicates that new parts of the program have been executed. Retranslating with the new profile will cause these additional parts of the image to be translated.

Alignment Issues

On an Alpha system, references to memory locations that are not naturally aligned result in exceptions that are handled by the Windows NT kernel. Alignment exceptions can be avoided by using unaligned code sequences that use the LDQ_U and STQ_U instructions. Unaligned code sequences are slower than aligned sequences for accessing locations that are naturally aligned but much faster for accessing locations that are not naturally aligned. Native Alpha compilers always try to generate unaligned code sequences when referencing unaligned data to avoid the expense of dealing with alignment exceptions.

When generating the code for an instruction that references memory, the code selector must determine whether to use an aligned sequence or an unaligned sequence. To make the determination, the code selector needs to know the alignment of the address being referenced. In general, this cannot be determined by static analysis of the x86 code. To solve the problem, the code selector uses information in the profile about the alignment of memory addresses. The profile contains the address of every instruction that made an unaligned reference to memory. The code selector generates unaligned sequences for those instructions and aligned sequences for all other memory references. Although this code generation process is effective most of the time, some programs exhibit different memory reference behavior on successive runs. For those programs, alignment exceptions can still occur.

Shadow Stack

Translating return instructions presented particular problems for the translator. The translation of a call instruction saves the x86 return address on the x86 stack and then calls the translated code for the routine. After the translated call, the x86 return address is on the x86 stack and the corresponding native return address is in an Alpha register. This maintains the x86 stack in the expected x86 state. One way to translate a return instruction would be to use the x86 return address to look up a corresponding Alpha address; however, it is desirable to avoid the expense of a hash table lookup on every return. In the usual case, the return address is not changed by the routine and the translated code can pop the x86 stack and perform a native return by using the native return address. Two

problems must be solved, though. First, some mechanism is needed to determine if the x86 return address has been modified. Second, a location is needed to save the native return address. Both problems are solved by using the shadow stack.

The shadow stack resides at the top of the native Alpha stack and is maintained by the translated code (with help from the emulator). A shadow stack frame is created for each call of a translated routine. When one translated routine calls another, the calling routine saves the x86 return address and the current x86 stack pointer in its shadow stack frame. The called routine then saves the native return address in the calling routine's shadow stack frame. On return, the called routine expects to find the x86 return address and the current x86 stack pointer in the calling routine's shadow stack frame. In this case, the called routine is returning to the environment that the calling routine expected and performs a native return. If the value of either the return address or the stack pointer has changed from the value expected by the calling routine, the called routine returns to the emulator.

In a similar manner, the emulator uses the information in the shadow stack to determine when it can return to translated code. A number of conditions can cause translated code to reenter the emulator. For example, the emulator is entered if the target of a translated indirect jump instruction is not known at translation time. Having the emulator return to translated code on a return instruction minimizes the amount of time that is spent in the emulator; however, the emulator can only return to the translated code if it knows that it has a valid return address. The shadow stack provides a mechanism to perform that validation.

The Database

The database consists of two parts. As described for the runtime, the first part of the database is a directory tree that contains profile files, translator log files, and translated images. The second part of the database is kept in the registry and consists of information about x86 applications and images that the DIGITAL FX!32 software has run on the system, together with configuration information. The configuration information includes the maximum amount of disk space that can be used by FX!32, the maximum number of images that can be stored in the database, the default translation options, the work list that the server uses to schedule translations, and the DatabaseDirectoryList. The DatabaseDirectoryList is a list of paths to additional databases that are to be searched for image profiles and translation results when the image is first executed. Directories on this list can be used to access information about the image from other machines on a network, making available to a user translations performed on another, perhaps more powerful, machine.

The Server

The server is a Windows NT service that normally starts whenever the system is rebooted. The server automatically runs the translator when appropriate, thus making the translation process completely transparent to the user. The server also maintains the database to control DIGITAL FX!32 resource usage.

The Manager

Usually the operation of DIGITAL FX!32 software is completely transparent to the user. Like any other program, though, FX!32 consumes system resources and a user must be able to control that resource usage. One of the roles of the manager is to provide a user interface to the configuration information kept in the database.

Figure 2 shows the manager window. The upper pane contains information about the various applications that have been run on the system: the total amount of disk space being used for profiles and translations of images loaded by the application, the number of times the application has been run, the date when it was last run, and the optimizer (translator) status. The lower pane contains information about the images that have been loaded by the highlighted application in the upper pane: the total amount of disk space used to store the profile and translation of the image, the number of times the image has been loaded, the date on which it was last loaded, and the status of the last translation of the image.

By interacting with the manager, the user can control various aspects of FX!32 operation, such as the maximum amount of disk space to use, which information to retain in the database, and when the translator should run.

Results

The DIGITAL FX!32 development team had two primary goals for the software: (1) to achieve transparent execution of 32-bit x86 applications and (2) to yield approximately the same performance as a high-end x86 platform when running applications on a high-performance Alpha system. The DIGITAL FX!32 product meets both goals.

Transparency is provided by the transparency agent and a run-time environment that can load and execute an x86 application without a translation step. Applications can be launched and executed on an Alpha system that is running FX!32 just as they can on an x86 system. We have performed extensive testing of more than 75 applications that run using FX!32, including major commercial applications such as Microsoft Office 95, Visual Basic 4.0, Photoshop 4.0, and CorelDRAW 6.0.

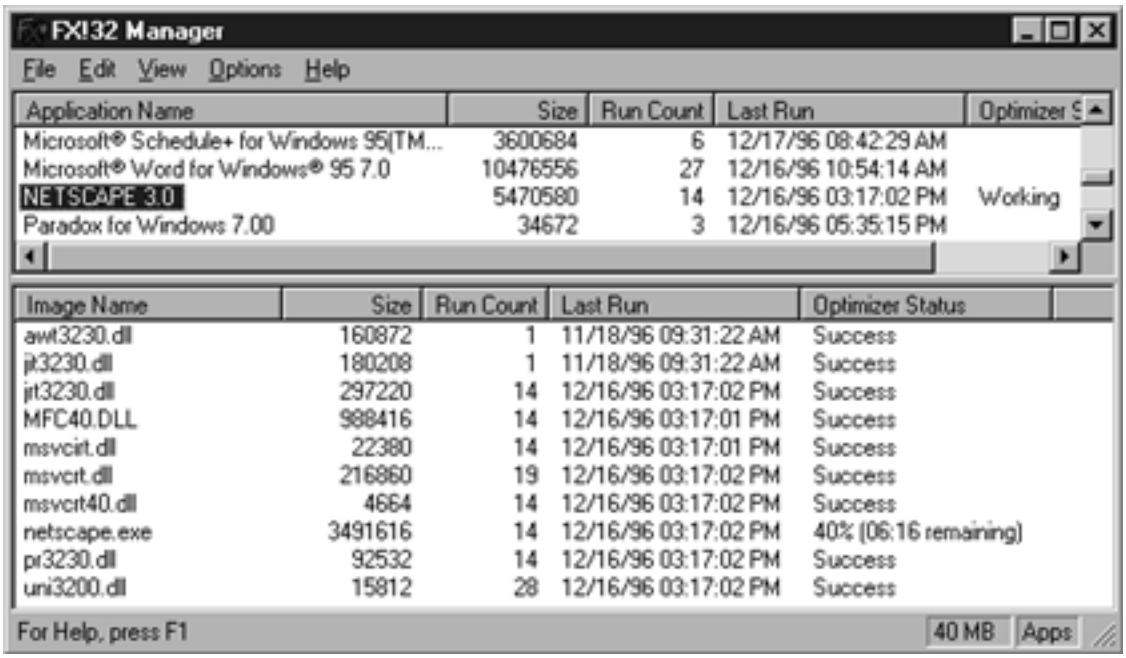

Figure 2
The DIGITAL FX!32 Manager

DIGITAL FX!32 software also met its performance goal. Figure 3 shows the relative performance on *BYTE Magazine*'s BYTEmark benchmark of a 200-megahertz (MHz) Pentium Pro system and a 500-MHz Alpha system running FX!32. For this benchmark, the Alpha system provides about the same performance as the 200-MHz Pentium Pro system. Figure 3 also shows that the Alpha native version of the benchmark runs twice as fast as the Pentium Pro version.

Of course, no single benchmark characterizes the performance of a system. Even so, when running translated x86 applications, we have consistently measured performance on a 500-MHz Alpha system to be in the range between that of a 200-MHz Pentium system and that of a 200-MHz Pentium Pro system. For

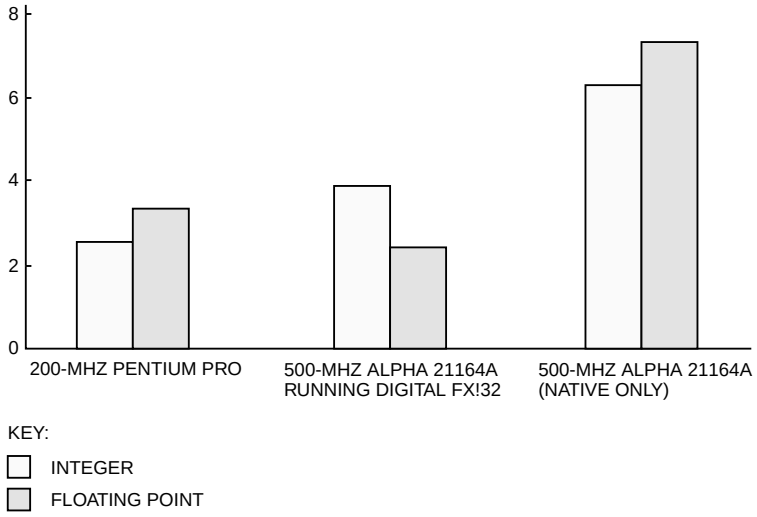

Figure 3
DIGITAL FX!32 Performance on the BYTE Benchmark)

some applications, performance can exceed that of a Pentium Pro system.

The initial version of the DIGITAL FX!32 software has some limitations. FX!32 executes only application code; it does not execute drivers. Consequently, native drivers are required for any peripheral that is installed on an Alpha system. Also, as described in the Transparency Agent section, FX!32 does not provide complete support for x86 services. Further, FX!32 does not support the Windows NT Debug API. Supporting that interface would require the capability to rematerialize the x86 state after every x86 instruction, thus severely limiting optimizations that the translator could perform. Optimizing compilers make a similar trade-off by restricting optimization when debugging information is required. Since FX!32 does not support the Debug interface, applications that require it do not run under FX!32. Those applications are mostly x86 development environments, and it probably makes more sense to run them on an x86 system. The limitations described are not serious, and most x86 applications that execute on an x86 processor that is running the Windows NT operating system also execute on an Alpha system running Windows NT and DIGITAL FX!32 software.

Summary

DIGITAL FX!32 software provides fast, transparent execution of 32-bit x86 applications on Alpha systems running the Windows NT operating system. This is accomplished using a unique combination of emulation and binary translation. The emulator runs an application, interprets the code, and generates profile information. For subsequent executions, the translator uses the profile data to produce translated images that contain optimized native Alpha code. An application translated by means of DIGITAL FX!32 software runs up to 10 times faster than the same application running under the emulator alone. Moreover, the translation takes place in the background and is therefore transparent to the user.

Acknowledgments

Building the DIGITAL FX!32 product required some extremely talented people to perform a lot of difficult work. The members of the DIGITAL FX!32 development team include Jim Campbell, Anton Chernoff, George Darcy, Tom Evans, Jim Givler, Charlie Greenman, Pippa Jollie, Mark Herdeg, Ray Hookway, Maurice Marks, Srinivasan Murari, Brian Nelson, Scott Robinson, Norm Rubin, Sherry Seskavich, Joyce Spencer, Tony Tye, and John Yates. Many of these individuals contributed the ideas described in this paper.

References

1. B. Case, "Rehosting Binary Code for Software Portability," Microprocessor Report (Sebastopol, Calif.: MicroDesign Resources, January 1989).
2. T. Halfhill, "Emulation: RISC's Secret Weapon," *BYTE Magazine* (April 1994).
3. R. Bedichek, "Some Efficient Architecture Simulation Techniques," *USENIX* (Winter 1990).
4. L. Deutsch and A. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages* (1983).
5. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," *Digital Technical Journal,* vol. 4, no. 4 (Maynard, Mass.: Digital Equipment Corporation, 1992).
6. J. Richter, *Advanced Windows NT,* chap. 16 (Redmond, Wash.: Microsoft Press, 1994).
7. A. Aho, M. Ganapathi, and S. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM Transactions on Programming Languages and Systems,* vol. 11, no. 4 (October 1989).

Biographies

Raymond J. Hookway
Ray Hookway led the DIGITAL FX!32 development team and was a key contributor to the binary translation component of the DIGITAL FX!32 software product. He has been a member of the AMT group of DIGITAL Semiconductor since 1993. Ray joined DIGITAL in 1989 and has worked in the CAD and AD groups of DIGITAL Semiconductor, where he contributed to the first Alpha PC project. Prior to joining DIGITAL , he was Director of Engineering for Endot, Inc., where he developed one of the first VHDL simulation environments. He was also an Assistant Professor at Case Western Reserve University, where he did research on program verification, and he was a Visiting Professor at the University of Upsalla, Sweden. Ray received M.S. and Ph.D. degrees in computer science from Case Western Reserve University and a B.S. in engineering from Case Institute of Technology. He has applied for several patents related to his DIGITAL FX!32 work.

Mark A. Herdeg
Mark Herdeg has been with DIGITAL since 1985. He is currently a principal software engineer in the AMT group of DIGITAL Semiconductor. Previously, he worked on console software for the Nautilus (VAX 8500) and Argonaut projects. The Alpha simulator developed for the Argonaut project, MANNEQUIN, became the first Alpha system on which the OpenVMS operating system successfully booted. Mark contributed to a related project that used the Alpha simulator and a dual–architecture-aware debugger to allow development and execution of applications with a mix of VAX and Alpha code. A founding member of the Alpha Migration Tools group, Mark worked on its first product, VEST, the OpenVMS VAX–to–Alpha binary translator. He then helped design and develop the DIGITAL FX!32 software product, with particular focus on the runtime component. Currently, he is the project leader for the next release of DIGITAL FX!32 software. Mark has submitted several patent applications for work on the multiple-architecture execution environment and for the DIGITAL FX!32 design.