FX!32

A Profile-Directed Binary Translator

Anton Chernoff

Mark Herdeg

Ray Hookway

Chris Reeve

Norman Rubin

Tony Tye

S. Bharadwaj Yadavalli

Digital Equipment Corporation

John Yates

Chromatic Research

A new innovation from Digital allows most x86 Windows applications to run on Alpha platforms with good performance.

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However, many other applications are available only under the x86 architecture. We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha. The goal for the software is to provide fast and transparent execution of x86 Win32 applications on Alpha systems. FX!32 achieves its goal by transparently running those applications at speeds comparable to high-performance x86 platforms. Digital FX!32 is a software utility that enables x86 Win32 applications to be run on Windows NT/Alpha platforms. Once FX!32 has been installed, almost all x86 applications can be run on Alpha without special commands and with excellent performance.

Before the introduction of this software, two common techniques for running an application on a different architecture than the one for which it was originally compiled were emulation and binary translation. Each technique has an advantage, but also a drawback. Emulation is transparent and robust, but delivers only modest performance. Binary translation[1] is fast, but not transparent. For the first time, Digital FX!32 combines these technologies to provide both fast and transparent execution.

This software consists of three interoperating components. There is a runtime environment providing transparent execution, a binary translator (the background optimizer) providing high performance, and a server coordinating them. Although FX!32 is transparent and does not require user intervention, it includes a graphical interface for monitoring status and managing system resources.

The first time an x86 application runs, all of the application is emulated. Together with transparently running the application, the emulator generates an execution profile describing the application's execution history. The profile shows which parts of the application are heavily used (for each user) and which parts are unimportant or rarely used. While the first run may be slow, it "primes the pump" for additional processing. Later, after the application exits, the profile data directs the background optimizer to generate native Alpha code to replace all the frequently executed procedures. The next time the application runs, native Alpha code is used and the application executes much faster. This process repeats whenever a sufficiently enlarged profile shows that it is warranted.

Three significant innovations of Digital FX!32 include transparent operation, interface to native APIs, and, most importantly, profile-directed binary translation.

Transparent operation

When we say FX!32 is transparent, we mean two things: applications execute in the expected way (without any special commands), and interoperability with native applications works normally.

Launching x86 applications. Transparent launching of Win32 x86 applications comes from a dynamically linked library (DLL), the transparency agent. Launching an application on Windows NT always results in a call to the CreateProcess function. By intercepting calls to CreateProcess, the transparency agent can examine every image as it is about to be executed. If a call to CreateProcess specifies an x86 image, the transparency agent instead invokes the FX!32 runtime to execute the image. Although special privileges are required to install FX!32, once installed, the transparency agent, and therefore the applications themselves, run without special privileges.

Digital FX!32 inserts the transparency

0272-1732/98/$10.00 © 1998 IEEE

agent into the address space of each process. A process containing the transparency agent is said to be enabled. Once a process is enabled, any attempt to execute a Win32 x86 image causes the runtime to start executing the process. The agent propagates 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, all top-level processes have been enabled by Digital FX!32, and any attempt to execute a Win32 x86 application invokes FX!32's runtime.

Processes are enabled by injecting a copy of the transparency agent into the process's address space, using a technique described by Richter.[2] The transparency agent's initialization routine then modifies a number of imported entry points by changing the addresses in the image import tables of all loaded modules to point to routines in the agent.

The transparency agent provides a general mechanism to change the behavior of an API routine called from Alpha code. We use this in a number of ways. For example, the behavior of the Win32 API routine LoadLibrary changes so that FX!32 loads x86 images. This is important because an attempt to load an x86 image on an NT Alpha system using the native loader results in an error. As another part of its function, FX!32 jackets the x86 image's exports so that they can be called from native Alpha code (discussed later). Finally, if FX!32's runtime is not already in memory, the transparency agent loads the runtime when it loads an x86 image.

The transparency agent we developed can be used for utilities besides Digital FX!32. For example, the transparency agent supports SPIKE (once known as OM), an Alpha native link-time optimization tool.[3] Users of SPIKE need only mark an application as interesting and every internally used library and image will be translated.

Runtime environment. The Windows NT operating system invokes the FX!32 runtime via the transparency agent whenever the user runs an x86 Win32 application. The runtime provides transparent execution because it contains an emulator that implements the entire x86 user-mode instruction set, and because it supports the complete x86 Win32 environment.

When the application first executes, Digital FX!32 has no knowledge of this application for this user and so runs it completely in the emulator. (As explained later, application calls to the x86 Win32 APIs, in fact, call corresponding native Alpha APIs.) The next execution of the application runs translated code for greater performance. The emulator remains present to interpret those x86 instructions that, for whatever reason, cannot be translated.

The rest of the transparency is provided by full support for the Win32 environment, such as multiple threads, structured exception handling, and the Microsoft component object model (COM) architecture across both the Alpha and x86 architectures. The runtime allows interfaces to all COM objects to be accessed from either x86 or Alpha code.

Runtime operation

The performance of Digital FX!32 comes from executing high-speed, native Alpha code. To secure high performance, the runtime transparently substitutes native Alpha code in place of x86 code whenever possible.

The FX!32 runtime is invoked whenever an enabled process attempts to execute an x86 image. The runtime loads the image into memory, sets up the runtime environment, and then starts emulating the image.

The runtime loader duplicates the functionality of the NT loader. This is necessary since the Alpha NT loader returns an error indicating that the image is of the wrong architecture if the loader is invoked to load an x86 image. This would have been much simpler had we been able to modify NT. Duplicating the functionality of the NT loader requires that the runtime relocate images not loaded at their preferred base address, set up shared sections, and process static thread local storage (TLS) sections.

After the image is loaded, the loader inserts pointers to the image into various lists used internally by NT. Maintaining those lists allows the native Windows NT code to treat both x86 and Alpha images identically. Fortunately, those image lists are in the user's address space, and no modification of NT is required. Unfortunately, the structure of those lists is not part of the documented Win32 interface and using them creates a dependency on the version of NT being run. This is one of a number of places where Digital FX!32 has dependencies on undocumented NT features, making it more dependent on a particular version of NT than a typical layered application would be. On the other hand, it is remarkable that Digital FX!32 implementation required no changes to NT.

Next, the image is entered into FX!32's database. The database provides the name of the translated image to be used with a given x86 image. The database is accessed by using an image ID obtained by hashing the image's header. The ID uniquely identifies the image by its contents, independent of the image's name or location in the file system. Both the runtime and the server use the image ID to access information stored in the database about the image.

If there is a translated image in the database, the runtime

Availability

Digital FX!32 is available free electronically from http://www.service.digital.com/fx32/.

This Web site contains more information on Digital FX!32 and the software itself. Additional information is available in Chernoff and Hookway,[1] Hookway and Herdeg,[2] and Thompson.[3] Digital FX!32 is a trademark of the Digital Equipment Corporation. All other trademarks and registered trademarks are the property of their respective owners.

1. A. Chernoff and R. Hookway, "Running 32-Bit x86 Applications on Alpha NT," *Proc. USENIX Windows NT Workshop*, Usenix Assoc., Sunset Beach, Calif., 1997, pp. 17-23.
2. R. Hookway and M. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," *Digital Tech. J.*, Vol. 9, No. 1, 1997, pp. 3-12.
3. T. Thompson, "An Alpha in PC Clothing," *BYTE*, Vol. 19, No. 2, Feb. 1996, pp. 195-196.

loads that image along with the original x86 image. Translated images are normal NT DLLs loaded by the native loader. A translated image contains the translated Alpha code, together with the two additional sections that define the correspondence between x86 code and Alpha code:

- A section containing relocation information for references to the x86 image. If the x86 image was not loaded at its preferred base address, those references must be relocated.
- A section containing a map between x86 and translated routine entry points. The runtime processes this map to update a hash table, indexed by x86 addresses, with entries pointing to the corresponding translated code.

Once the images are loaded, the runtime starts emulating the x86 instructions. When the emulator interprets a CALL instruction, it looks for the target x86 address in the hash table. If a corresponding translated address exists, the emulator transfers to the translated code. The emulator also generates profile data for use by the translator containing the following information:

- addresses that are targets of CALL instructions,
- source address/target address pairs for indirect jumps, and
- addresses of instructions that make unaligned references to memory.

The profile data is collected by inserting values into the runtime hash table whenever a relevant instruction is emulated. For example, when emulating the CALL instruction, the emulator records the call's target. When an image is unloaded, or when the application exits, the hash table is processed, and a profile for that image is written. The server processes this profile, merging it with any previous profiles and may invoke the translator.

Cross-architecture interoperability

Win32 applications make calls to routines that are not part of the application, specifically the Win32 API. Because these are x86 applications, they make calls by using the x86 calling conventions. NT Alpha provides the same routines, but with Alpha calling conventions. FX!32 provides a mechanism to connect the two.

Transformations are required to manage a call between a native Alpha routine and a piece of emulated or translated code. For example, x86 routines pass arguments on the stack while Alpha routines expect arguments in registers. Small code fragments called jackets, which manage the transition between the x86 and Alpha environments and calling conventions, perform these transformations.

There are two basic kinds of jackets, static and dynamic, based on how and when they are created. Static jackets are created from a defined interface known at load time. They are included as part of Digital FX!32's runtime. Most static jackets are simple and are generated automatically from documentation and header files. Some static jackets are built by hand because code is required to process the arguments in a special way. Digital FX!32 provides static jackets for the Win32 API interface, NT call-back routines, standard object linking and embedding (OLE) objects, and some selected plug-in extensions.

COM objects whose interfaces are not statically available are dynamically jacketed at runtime. These dynamic jackets are created by using type information obtained from the OLE libraries.

Interface to native APIs

Unlike Unix, in Windows NT most system APIs are part of the operating system. For example, most graphical user interface functions are built into NT system DLLs. We found that some applications, such as Microsoft Excel, spend almost half their execution time in these libraries. We knew that it was very important for Digital FX!32 to call native libraries whenever possible to achieve our performance goals.

When the NT loader loads an image, the loader "snaps" the image's imports by using symbolic information in the image to locate the addresses of the imported routines or data. The runtime duplicates this process. However, the runtime treats imports referring to entries in Alpha images specially, by redirecting them to refer to the correct jacket entry.

Each jacket contains a special illegal x86 instruction that serves as a signal to the emulator to switch into the Alpha environment by calling Alpha code 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. For example, the jacket for GetSystemDirectory returns the path to the x86 system directory rather than to the true system directory, so x86 applications do not overwrite native Alpha DLLs.

Jacketing the Win32 API. Previous translation utilities (for various Unix flavors) created by Digital jacketed the operating system call interface because that was the defined interface between applications and the operating system. This required jacketing an interface to about 100 system calls. Windows NT defines and documents the Win32 API (layered above the system call interface) as the interface between applications and the operating system, and Digital FX!32 jackets the complete Win32 API. Although jacketing the complete Win32 API is a significant task, it is required to guarantee correctness and provides better initial performance because the jacketed routines are native and do not need translation. As a result, Digital FX!32 provides static jackets for entries to over 50 native Alpha DLLs, including jackets for many undocumented routines. About 12,000 routines are currently jacketed.

Jacketing call-back routines. Many Windows NT routines are passed the addresses of routines to call back when an event occurs. If these values were to be passed blindly, the Windows NT Alpha code would make a call to a location containing x86 code and would certainly crash. A jacket is statically created for each procedure-pointer argument, and the address of that jacket is passed to the native Alpha code. When Alpha code calls back to its argument, the jacket enters the FX!32 runtime.

Jacketing COM objects. The most complicated jacketing

problem is associated with COM. A COM object is represented by a table of OLE function pointers. These functions often have arguments that are pointers to functions or structures containing pointers to functions. Digital FX!32 manages these objects in a way that can be used from either native Alpha or x86 code.

Jacketing plug-in extensions. For full interoperability, it is also desirable to support x86 plug-ins (add-ons or extensions defined by an application vendor) with the corresponding native application, when it is available. Each of these introduces another interface (requiring jackets) that is not defined by NT and not available at runtime. Digital FX!32 cannot load such a plug-in unless it is programmed to jacket the interfaces. The current version of Digital FX!32 jackets a few common plug-in interfaces—we are working on ways to describe arbitrary plug-in interfaces for a future release.

Runtime and background optimizer

Commercial applications typically consist of numerous executable files, called images. Some images are unique to the application, and some are shared across different applications on the system. Each time the runtime loads an x86 image, the runtime queries the database as to whether translated code exists for that image to run in place of the slower x86 code. Translated code is high-speed, native Alpha code, produced by the background optimizer after previously emulating the image under Digital FX!32.

After loading the translated code, the runtime sets up tables that correlate addresses between any x86 code and the translated code. The runtime then initiates the emulator, which starts executing the application. From careful design and alignment with the Alpha architecture, the emulator is both small and efficient. The emulator is small enough to reside mostly in the high-speed instruction cache, is optimized for the Alpha processor pipeline, and takes full advantage of the 64-bit Alpha processor registers.

As it emulates untranslated portions of x86 images, the runtime collects and saves execution profiles for subsequent use by the background optimizer. The performance of Digital FX!32 is based on this cooperation between the runtime and the background optimizer.

Coordinating the process: the server

The server manages FX!32's environment by coordinating the runtime and the background optimizer. The server acts according to Digital FX!32 defaults or according to parameters that can be specified by the user. In response to these parameters, the server manages execution profiles and invokes the background optimizer.

After an x86 image is unloaded, the server merges any new profile information with any existing profiles and compares the size of the result with any previous size. A new profile means that a previously unseen x86 image has been executed and may require optimization. An enlarged profile contains new information, indicating that the current optimized image is incomplete. In either case, the server places the image and the corresponding profile on the work list for the background optimizer.

This process is repeated each time the image runs. Figure 1 shows the execution flow among FX!32 components. When the size of the profile stabilizes (typically at two or three iterations), it indicates that virtually all executed routines in the image are translated. The image and corresponding profile are no longer placed on the work list for the background optimizer. Running the image executes high-performance, native Alpha code, rather than the slower x86 code. The image runs at its highest performance.

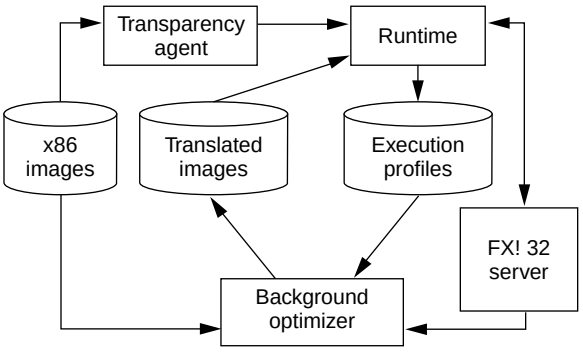


Figure 1. The flow of information among Digital FX!32 components.

Creating the speed: binary translation

The background optimizer, a third-generation, profile-directed binary translator, produces high-speed, native Alpha code from x86 code by using information gathered into profiles by the runtime. A binary translator is a program that, from the original code, produces translated native code that can be executed directly. The native Alpha code is subsequently made available to the runtime and executed the next time the image is run. It is this coordinated process that adds high performance to the transparency of execution.

Design goals. The operation and output of the background optimizer must be as transparent and robust as the runtime environment. The user never sees the operation of the background optimizer; it always presents code to the runtime that runs to correct completion. To ensure transparency, the background optimizer design allows for no assumptions, no manual initiation, and no user intervention in any question/answer cycle.

Coupled with the stringent need for transparent and flawless operation is a requirement for the highest possible performance.

Realization of the goals. The background optimizer guarantees transparent and robust operation by cooperating with the runtime to ensure a faithful representation of the x86 machine state. A coherent x86 machine state means the x86 register assignments, call/return boundaries, and the x86 stack all reflect what would be observed on actual x86 hardware at relevant observation points.

Achieving the performance goals required us to exploit the full range of modern compiler optimization techniques, which are all predicated on global optimization.

Previous binary translators operated with a poor quality

Digital FX!32 is the first system to exploit this combination of emulation, profile generation, and binary translation.

approximation of the application's control flow graph. As a consequence, they were limited to the basic block, or perhaps the extended basic block, as the fundamental unit of translation. (A basic block is a sequence of instructions with a single entry point and a single exit point.) All modern optimizing compilers require global optimization techniques that directly conflict with such a basic-block unit restriction. Therefore, removal of this restriction was the fundamental performance requirement. The background optimizer successfully removes this restriction by using profiles to organize carefully chosen groupings of basic blocks into significantly larger units, called translation units. Conceptually, a translation unit approximates a "routine" in a more traditional compiler and thus allows the full exploitation of global optimization techniques.

Profile-directed binary translators

Digital has used other binary translation techniques in the past,[1] mainly static binary translation. Our development group has extensive experience with previous binary translators. We also looked at other solutions, such as hardware engines and dynamic binary translation.

In Digital FX!32, we have developed a new approach to translation. The emulator captures an execution profile, which the binary translator subsequently uses to translate executed x86 code into native Alpha code. Since the translator runs in the background, it can use complex algorithms to improve the quality of the generated code. To our knowledge, Digital FX!32 is the first system to exploit this combination of emulation, profile generation, and binary translation. We call our approach profile-directed to contrast it with static and dynamic approaches.

Because we have the execution profile, our binary translator was easier to write, runs faster, and produces better code than any previous static binary translators. Our translator was easier to write because the complex search algorithms and heuristics used to find the code and the control flow graph were replaced by much faster and simpler lookups. Digital FX!32 produces better code because the profiles result in more accurate approximations of the control flow graph, allowing optimizations to be more effective.

Translator operation

In many ways our binary translator is a traditional high-performance compiler. However, there is an important difference. Compilers start from source level and proceed to lower the semantic level, while binary translators start with bits and raise the semantic level first to instructions and then to control flow graphs. The challenge for the translator is to produce correct and efficient code in this framework.

Locating code. The search for code begins at all the destinations of call instructions recorded in the profile. As the code is parsed, the destinations of indirect branches are resolved by looking in the profile. As a consequence, no complex and slow iterated data flow is required. The profile-directed approach needs less code in the translator even though this approach makes a more accurate determination of the location of code and control flow edges.

Since the translator builds a good approximation to the control flow graph, basic blocks can be joined into larger units. The translator contains a component called the regionizer that divides the x86 image into routines.[4] Routines are units of translation that approximate real routines in source programs.

The regionizer represents routines as a collection of regions. Each region is a contiguous range of addresses containing instructions that can be reached from an entry address of the routine. Routines end at the return statements identified by the profile. Unlike basic blocks, regions can have multiple entry points. The smallest collection of regions containing all the instructions reachable from the routine entry represents the routine. Most routines have a single region. This representation efficiently describes the division of the source image into units of translation.

Intermediate representation. The remaining translator components process the source image one routine at a time. All control flow is explicitly represented (including all the direct control flow), as well as indirect control flow recorded in the profile information. For every transfer of control that might have additional unknown destinations (such as indirect branches), the translator inserts a call to the emulator. Only the routine's entry points are entered in the x86-to-Alpha correlation table, ensuring that the emulator cannot transfer to an arbitrary block in the routine.

The emulator and translator share a canonical representation of the x86 state. In the translator, all entries into and out of the routine use explicit intermediate representation to represent the canonical x86 state. Other than at these points, the translator is free to use whatever representations for the x86 state it finds convenient. As a result, the transformations and optimizations do not have to be as conservative as in the static translators, which have to allow for the emulator transferring control to almost any basic block. This allows the translator to perform global transformations and optimizations on the whole routine.

A more accurate control flow graph, based on the profile, is vital to our performance. Each time the application executes, an indirect branch to a target not previously executed invokes the emulator. Once in the emulator, translated code is not resumed until another routine is called or the routine returns to a translated caller. The runtime then adds that fact to the profile.

The same intermediate representation has primitives for both x86 and Alpha operations. The processing of a routine starts by building a representation of the x86 code. Then, multiple transformations convert the representation from an x86 semantic model to an Alpha semantic model. Optimiza-

tion phases are interspersed with these transformations. At the end of processing each routine, the final Alpha code is assembled into the translated image.

Translation and optimization

Our goal was to handle a very large percentage of x86 applications, including those that do not follow the NT calling conventions. We knew that Digital FX!32 would need to maintain great fidelity. The translator uses a simple code generator to map x86 instructions into a correct general, but long, sequence of Alpha instructions. Then the translator uses global transformations and optimization to improve the code.

The translator uses many traditional compiler techniques. It includes optimization phases for dead-code elimination, constant propagation, common subexpression elimination, register renaming, global register allocation, instruction scheduling, and numerous peephole optimizations.

Condition code management. Most x86 instructions generate condition codes, but only rarely are they consumed. Initially, the x86 model is represented in the intermediate representation with condition code information for each instruction. Global data flow determines the lifetimes of the x86 condition codes. Explicit Alpha code is then inserted to compute only those condition codes used.

Register management. The x86 architecture uses distinct registers to access different bytes of the same underlying register. The mapping of these overlaid registers to Alpha registers uses data flow to minimize the amount of generated Alpha code. Since the x86 state only has to be canonical at routine boundaries, the various overlays of an x86 register within a routine can be maintained in separate Alpha registers to allow more efficient access. This also allows global renaming to reduce register dependencies, increasing the benefits of instruction scheduling.

Stack management. The x86 architecture has few registers, so x86 code tends to make extensive use of the x86 stack to hold temporary results. The translator analyzes memory accesses to identify storing and loading from the x86 stack. The translator assumes that when the x86 stack is popped, any data stored above the new stack pointer is dead. The translator uses this information to eliminate those unnecessary loads and stores. Any loads and stores that cannot be proved to be unaliased are not eliminated. After eliminating loads and stores, the translator coalesces increments and decrements to the x86 stack pointer to minimize the number of updates, while preserving the runtime convention that the stack is never accessed above the stack pointer.

Routine management. We have found x86 routines that walk up the stack and modify local variables of their callers, including return addresses. To make these routines work, FX!32 needs to make the x86 application see an identical stack image. 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 native return address that corresponds to the x86 return address is in an Alpha register. In the usual case, the routine does not change the return address, and the translated code can pop the x86 stack and perform a native return by using the native return address. However, there are two problems to solve. First, it must be possible to determine whether the application modified the x86 return address. Second, there must be a place to save the native return address. Both problems are solved using the shadow stack.

The shadow stack resides at the top of the native Alpha stack and is maintained by the translated code and the emulator. A shadow stack frame holds the x86 and the Alpha return addresses, along with the x86 stack pointer at the time of the call. The translated code for a RET instruction uses these values to determine when it is not legal to make a native return, at which point the emulator is entered to start emulating from the modified x86 return address. The emulator consults the shadow stack when emulating RET instructions to see if translated code can be resumed. In this case, the emulator uses the Alpha return address in the shadow stack.

The emulator uses the x86 stack pointer saved in the shadow stack to remove shadow-stack frames above the current value of the x86 stack pointer. Such frames can occur if the code cuts back the x86 stack to return to an earlier caller (as is done by the longjmp C library routine). This cleanup always finishes before the emulator uses the shadow stack, ensuring the shadow stack does not overflow.

Alternative solutions

As mentioned previously, our primary design goals for Digital FX!32 were transparency and high performance. Before arriving at the coordinated combination of emulation, profile generation, and profile-directed binary translation, we examined a range of alternative solutions.[5-7]

Hardware-based solutions. One approach would have been to design a new chip that supports both the Alpha and the x86 ISAs. Similar techniques exist in a number of designs. The most popular variations on this approach use a hybrid design known as a decoupled microarchitecture. This design combines a high-performance execution core with a sophisticated x86 instruction decoder. The decoder translates x86 instructions into simpler operations that execute more efficiently. This approach can generate quite good performance on applications written for the x86. Some examples of machines that use this approach are the AMD K6, Intel Pentium Pro, and NexGen Nx586. None of these machines expose the alternative instruction set architecture (ISA) to the user, and therefore they pay the penalty of being basically CISC designs (albeit with a RISC core). This limitation could be overcome with an x86/Alpha chip that exposes both ISAs. However, we felt that Digital FX!32 could achieve good performance for x86 applications by using a totally software-based solution, avoiding the complexity of including support for the x86 ISA in future Alpha chip designs.

Software-based solutions. There are two common software alternatives that also allow applications written for one ISA to execute on a different ISA—emulation and binary translation.

Emulators. These programs, at runtime, dynamically execute instructions written in the original ISA. Many systems have successfully used emulators to run applications on platforms for which they were not targeted.[8] The major advantage of emulators is transparency. The major drawback is poor performance. For example, our x86 emulator, which we care-

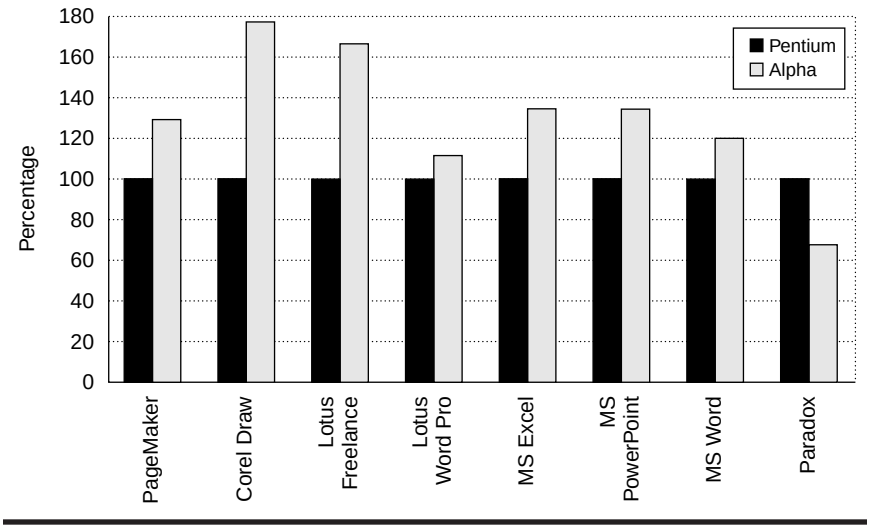


Figure 2. Relative Performance of a 500-MHz Alpha running Digital FX!32 and a 200-MHz Pentium.

fully wrote in Alpha assembler, requires an average of 45 Alpha instructions to emulate one x86 instruction (or 30 Alpha instructions per Pentium Pro micro-operation). While this is acceptable for infrequent use, it is too slow to meet our goals.

Emulators are commonly deployed in one of two ways: within a restricted environment or tightly integrated into the operating system. In a typical restricted environment, the user brings up an emulator window, and the emulator executes any application launched in that window. Our goal of transparent execution made us reject restricted environments.

In an integrated system, a modified operating system loader automatically launches the emulator whenever an emulated application is started. Windows NT has contained an emulator to run 16-bit x86 applications since it was first released on RISC platforms. Since we did not build Windows NT, we developed a scheme that allows us to launch emulated applications without needing source changes to NT.

Binary translators. These programs start with original code and produce translated native code that can be executed directly. The main advantage of binary translation is that the translated applications run at high speed. For example, after translation, Digital FX!32 executes an average of 4.4 Alpha instructions per x86 instruction (2.1 Alpha instructions per Pentium Pro micro-operation). Since the typical clock speed of an Alpha (500 to 600 MHz) is twice the clock speed of an x86 (166 to 233 MHz), it is clear that using a binary translator could achieve our goal.

Two previous types of binary translation already existed when we began to design Digital FX!32: dynamic translators and static translators.

Dynamic binary translators

Several emulators[9] have used dynamic translation, sometimes called just-in-time translation, or JIT, to achieve better performance. This approach translates small segments of an application while it is being executed. Systems using dynamic translation trade off the amount of time spent translating and the resulting benefit of the translation. Too much time spent on the translation and related processing makes the application unresponsive; too little time makes the performance slow.

Therefore, most of these systems limit the optimizations they perform to minimize the translation overhead. Dynamic translators are usually stateless, so that each time an application starts, the translator begins anew. For each application, the start of each execution serves as a training set that is then used to guide the dynamic translator. For code run only once, this is an attractive option. However, important applications are run repeatedly, and the initial training is thus repeated each time.

Static binary translators

The other existing software alternative is static translation. Here, a translator program scans the entire image and translates everything at once. We have built several static binary translators in the past, and developers and sophisticated end users have found them quite useful as a way to quickly port an application. Static binary translation is particularly useful when the source code for the application is not available or is prohibitively complex to recompile, as an interim solution while source code is being ported, or when the best possible performance is not an issue.

Static binary translator operation. The user manually invokes the translation tool to convert code from a non-Alpha ISA to Alpha. This scheme is difficult to use with an application that contains many images, because each image requires the user to manually invoke the tool. It is hard to get users to run tools that have many steps; users expect applications to "just work."

Static translators use a static approach to try to answer the following questions: what part of an image is code, what part is data, and what is the control flow graph?

Static translators separate an image into basic blocks using the following steps:

1. The static translator identifies a set of addresses considered to be the start of a basic block. It looks for addresses that meet the following criteria: they're externally visible in the text section of an image, the addresses serve as either an entry point or as the target of a relocated instruction, and they start a valid sequence of instructions that ends in a branch.
2. The static translator parses the identified basic blocks, finds the ending branch, and tries to determine the destination of the branch. Each such destination is considered the start of another basic block. For some branches, finding the destination is simple, but for others (such as indirect branches via a register), interprocedural global data flow is required. It is possible to identify a sequence of instructions as a single basic block and later find a

branch into the middle of that sequence. Thus the translator needs to iterate both the data flow calculations (to find possible values of registers) and the parsing of blocks (to find indirect branches).

3. Because the data flow calculation misses many possible values, the static translator walks over the text section and scans for missed basic blocks. It looks for any sequence of bits not part of a known basic block, but that could be parsed into a sequence of valid instructions ending in a branch.

At the end of this process, the static translator has a list of addresses in the source image that are likely to be the start of basic blocks, together with some control flow. As this sea of basic blocks is translated, the list of addresses expands into a structure called a correlation table. This table lists pairs that contain source machine addresses and the addresses of corresponding translated code.

At runtime, indirect branches are translated into a call to a library routine. This routine looks up the destination of the branch in the correlation table. If there is an entry, there is an available translation of the corresponding basic block and the library routine branches to the translation. If there is no entry, the library routine emulates up to the next branch and tries the lookup again.

Since the emulator can enter translated code at any block in the list of pairs, optimization is generally limited to single basic blocks. However, optimizations can be done across basic blocks, provided that the block is removed from the correlation table. Of course, any block disconnected from the control flow graph cannot be globally optimized.

Analysis of static binary translators. Although we were willing to use expensive techniques such as repeated full-image data flow, the static translators missed important control flow edges and sometimes saw edges that were never taken. The correlation table could contain entries for addresses that appeared to be the start of a block but were actually data. At the same time, the table could be missing entries for blocks reached by indirect branches.

The performance of static translations tends to depend upon how well destinations of indirect branches can be resolved. When we started to build Digital FX!32, we realized that the style of programming used in many x86 applications would make resolving these destinations very difficult. Static translators also provide no transparent way of executing an application, requiring a full translation was manually done before the application could be executed. This led us to consider a profile-directed translator in conjunction with an emulator that generates profiles.

What does not work?

The most obvious way in which Digital FX!32 is not transparent is that x86 applications are installed by using an add/remove x86 program applet visually and functionally similar to the NT add/remove program applet. Another nontransparency is that the first execution of an application is much slower than the second execution.

There are some things that the initial version of Digital FX!32 was not designed to do. Digital FX!32 only executes application code. It does not execute drivers, so a native driver is required for any peripheral device installed on an Alpha system. Digital FX!32 does not provide complete support for x86 NT services (services from the NT control panel services applet) because such services are enabled only when they are started after FX!32's server. We hope to remove this restriction in future versions of Digital FX!32.

Digital FX!32 does not support the NT debug API. Supporting that interface would require the ability to rematerialize the x86 state after every x86 instruction, severely limiting optimizations that could be performed by the translator. This limitation is similar to the trade-off in optimizing compilers where debugging is restricted when optimizations are turned on. Since Digital FX!32 does not support the debug interface, applications requiring it do not run under Digital FX!32. Those applications are mostly x86 development environments, and it probably makes sense to run them on an x86 anyway.

Performance

Figure 2 shows relative performance on a set of benchmarks for a 200-MHz Pentium and a 500-MHz Alpha with similar configurations. A larger number indicates higher performance. For the Alpha, we took the timings at the second execution of the benchmark using the same input data. For these benchmarks, the Alpha running Digital FX!32 provides roughly the same performance as a 200-MHz Pentium. These benchmarks are the set of applications included in the well-known PC benchmark, BapCo SysMark 32.

Of course, no small set of benchmarks characterizes the performance of a system. Even so, when executing translated x86 applications, we have consistently measured performance on a 500-MHz Alpha in the range between a 200-MHz Pentium and a 200-MHz Pentium Pro.

SINCE IT WAS FIRST RELEASED two years ago, Digital FX!32 has been used by thousands of NT/Alpha users, with over 13,000 copies downloaded from FX!32's Web site alone. At least five commercial redistributors of NT/Alpha systems have made FX!32 available on their own Web sites. FX!32 has also been factory-installed software on all NT/Alpha workstations shipped by Digital. Although, it's become the most widely used of all profile-directed software tools, development work remains. Specifically, FX!32's operation is still not completely transparent to the user. To install an x86 application on NT/Alpha, the user must check a box in the add/remove programs dialog box. Work remains to be done on the background optimizer so that its operation need not be scheduled, and the code produced by the optimizer is still not as close in performance to native Alpha code as we would like.

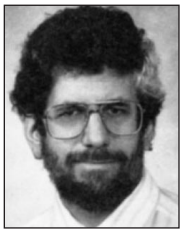
Acknowledgments

Building a product like Digital FX!32 required a lot of hard work by some extremely talented people. Many of these people contributed the ideas described in this article. The following engineers were part of Digital FX!32's development team: Rick Bagley, Jim Cambell, George Darcy, Paul Drongowski, Tom Evans, Charlie Greenman, Alan Krzywicki, Maurice Marks, Srinivasan Murari, Brian Nelson, Scott Robin-

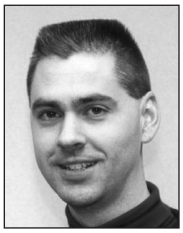
son, Joyce Spencer, John Strange, Monty Vanderbilt, and Andrew Wison.

References

1. R.L. Sites et al., "Binary Translation," *Digital Tech. J.,* Vol. 4, No. 4, 1992, pp. 137-152.
2. J. Richter, *Advanced Windows NT*, Microsoft Press, Redmond, Wash., 1994.
3. R. Cohn et al., "SPIKE: An Optimizer for Alpha/NT Executables," *Proc. USENIX Windows NT Workshop*, Usenix Assoc., Sunset Beach, Calif., 1997, pp. 9-15.
4. R. Hank and B.R. Rau, "Region-Based Compilation: An Introduction and Motivation," *Proc. MICRO-28,* IEEE Computer Society Press, Los Alamitos, Calif., 1995, pp. 158-168.
5. R. Bedichek, "Some Efficient Architecture Simulation Techniques," *Proc. USENIX Conf.,* Usenix Assoc., Sunset Beach, Calif., 1990, pp. 53-63.
6. R.F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proc. ACM Sigmetrics Conf.*, ACM Press, NY, 1994, pp. 128-137.
7. T.R. Halfhill, "Emulation: RISC's Secret Weapon," *BYTE*, Vol. 19, No. 4, Apr. 1994, pp. 119-130.
8. B. Case, "Rehosting Binary Code for Software Portability," *Microprocessor Report*, Vol. 3, No. 1, Jan. 1989, pp. 4-9.
9. L.P. Deutsch and A.M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proc. 11th Ann. Principles of Programming Languages*, 1983, pp. 297-302.

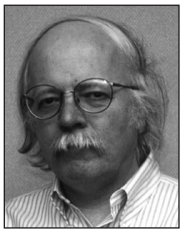


Anton Chernoff is a senior consulting engineer in the Alpha Migration Tools (AMT) group at Digital Equipment Corporation in Littleton, Massachusetts. He has been working on binary translation since 1991. Chernoff is an ACM member.

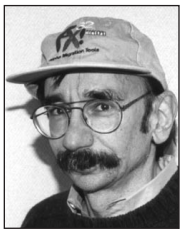


Mark Herdeg is a principal software engineer in the AMT group at Digital Equipment Corporation. He is a member of FX!32's development team, having contributed to its original design, and works primarily on the runtime environment. He has received several patents related to binary translation and emulation environments.

Ray Hookway is a consulting engineer in the AMT group at Digital Equipment Corporation. His interests include binary translation and compilation. He received MS and PhD degrees from Case Western Reserve University.

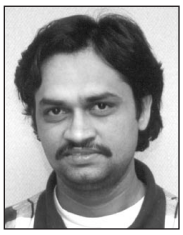


Chris Reeve is a member of the technical staff at Digital Equipment Corporation. He is a member of the FX!32 group working on the binary translator. His research interests include optimizing compilers for high-performance microprocessors and parallel processing. Reeve has BS and MS degrees from MIT.

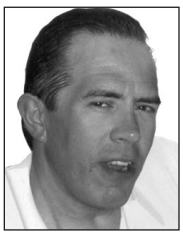


Norman Rubin is a member of the technical staff in the Advanced Products Unit in Digital Equipment Corporation's AMT group. Rubin's technical interests include optimizing compilers, profile-directed compilation, computer architecture, object-oriented programming, and Java. Rubin received his MS and PhD degrees in computer science from the Courant Institute of NYU. He is a member of both the IEEE and ACM.

Tony Tye is a software engineer in Digital Equipment Corporation's AMT group. He has been a member of FX!32's engineering team since its inception, primarily working on the offline optimizer component. Tye received his MS and PhD degrees in computation from the University of Manchester Institute of Science and Technology, England.



S. Bharadwaj Yadavalli is a software engineer in FX!32's AMT group at Digital Equipment Corporation. His current technical interests include binary translation, compiler optimizations, computer architecture, and related high-performance system tool development issues. Yadavalli obtained his MTech in computer science from Jawaharlal Nehru University, India, and is currently completing his PhD at University of Saskatchewan, Canada.



John Yates works on mass-market computing at Chromatic Research. He is happiest working at the hardware/software boundary, lapsing regularly into assembler (such as FX!32's x86 interpreter). He has designed and shipped a computer microarchitecture and instruction set, various runtime models, and numerous compiler components (including an SSA intermediate language).

Direct questions concerning this article to Norm Rubin, Digital Equipment Corporation, rubin@amt.tay1.dec.com.