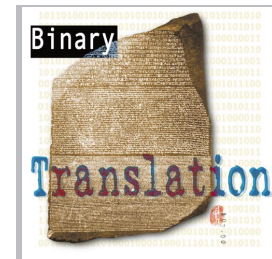# Welcome to the Opportunities of Binary Translation

A new processor architecture poses significant financial risk to hardware and software developers alike, so both have a vested interest in easily porting code from one processor to another. Binary translation offers solutions for automatically converting executable code to run on new architectures without recompiling the source code.

**Erik R. Altman**
IBM T.J. Watson Research

**David Kaeli**
Northeastern University

**Yaron Sheffer**
Radguard

Today's most commercially significant microprocessors remain firmly wedded to legacy instruction set architectures, some of which are now over a decade old. Despite the admitted shortcomings of these ISAs, manufacturers are reluctant to develop radically new ones because they risk losing the commercial advantage of their product's existing software base.

On the other side of the coin, software developers find porting code to a new architecture difficult and time-consuming. If the architecture fails to gain enough market share, they—like the hardware developers—also risk losing a significant investment.

Both these factors have at times conspired to forestall innovation in processor design. They also make it more difficult for new competitors to arise. *Binary translation*—a set of techniques that directly translate compiled code—could help break the innovation-strangling relation between ISAs and their software base.

## OPTIONS FOR A NEW ISA

Designing a new ISA is expensive, particularly because it often requires recompiling several operating systems and applications. Modern languages don't define their semantics tightly enough to make recompilation transparent. In contrast, the semantics of binary code is usually well defined, facilitating automatic and transparent translation. Developers have practiced binary translation for many years, but only with recent increases in processing power has it become possible to fully use translation.

When porting legacy code from a legacy ISA to a new architecture, you can

- provide a special processor mode to execute legacy code on the new processor,

- recompile the program to the new instruction set, or
- use a variety of software methods to interpret or translate the application.

We focus in this special issue on the third class, the software methods that let you translate at runtime or while offline. Each of the three articles that follow demonstrates a different approach to software translation:

- Hewlett-Packard's dynamic translator, Aries, eases the transition of applications from HP's Precision Architecture to IA-64,
- IBM Research's BOA translates the legacy PowerPC code of a full system rather than just the application code, and
- The University of Queensland's UQBT is a binary translation system that—instead of translating between a specific pair of instruction sets—lets you generalize binary translation and work between virtually any two architectures.

Figure 1 shows the universe of three various translations.

## THREE TYPES OF TRANSLATION

Software-based binary translation systems—including those discussed in the following articles—can be classified as emulators, dynamic translators, or static translators. An emulator interprets program instructions at runtime. The system doesn't save the interpreted instructions or cache them. Emulators are relatively easy to develop and with only modest effort can be made highly compatible with the legacy architecture.

A dynamic translator, on the other hand, translates between the legacy and the target ISA, caching the pieces of code for future use. Java JIT (just-in-time)

compilers are probably the best-known translators in this class. The sidebar "A Just-in-Time Compiler" describes Latte, a JIT compiler that generates high-quality Sparc code from Java bytecode.

A static translator translates programs offline and can apply more rigorous code optimizations than can a dynamic translator. Static translators can also use execution profiles obtained during a program's previous run.

All three approaches have limitations: Both emulation and dynamic translation impose runtime overhead, while static translation as a stand-alone tool requires end-user involvement. Much recent work in this area revolves around innovative hybrid solutions to combine the best features of each approach.

## RUNTIME OPTIMIZATION

For binary translation to serve as a viable alternative to legacy architecture execution, the performance of the translated target executable should be competitive with the legacy architecture's performance. The legacy architecture executable has the luxury of being produced using an optimizing compilation process.

Binary translation generally does not have high-level-language code available and thus works solely from the executable code. Not knowing the full semantics of the original source code, a binary translator cannot perform many optimizations available to a compiler.

### Profiling

One common approach to improving binary translation performance is profile-guided optimization. As the sidebar "Collaborative Profiling" shows, execution profiles can be generated efficiently. They are then used to guide optimization performed during the translation process and during further tuning of the translated image.

A static translation system can merge multiple profiles into a single profile. A dynamic translation system can retranslate translated code periodically as the profiles change.

### Dynamic optimization

Sometimes a dynamic translator forms part of the translated code's execution thread, which means that execution stalls can occur during translation. Dynamic translators often make use of program behavior information to optimize execution. Hewlett-Packard's Dynamo, discussed in the sidebar "Native Binary Acceleration," is a good example of dynamic optimization.

Other optimization techniques used in binary translation include:

- *ISA remapping* to handle register overlaps present in the legacy ISA and remap them to the target ISA;
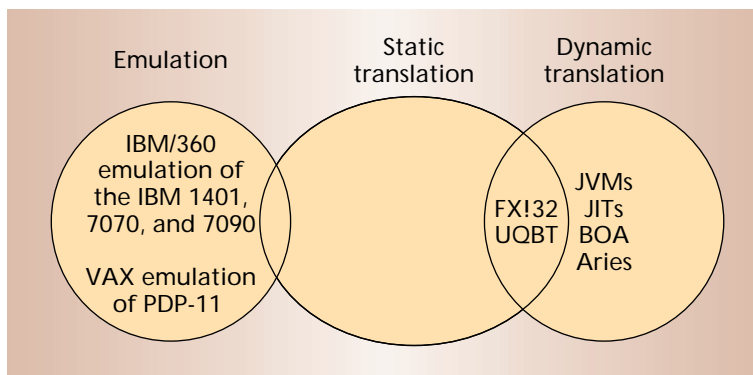


Figure 1. The universe of binary translators.

- *basic block reordering* to keep the target image execution as sequential as possible so that conditional branches will typically fall through, which helps speed instruction fetching and cache performance;
- *memory coloring* to improve the mapping of the translated image onto the memory hierarchy of the target environment;[1] and
- *code specialization* to clone procedures based on the invariance of parameter values.

Any optimization considered by the binary translation system can be used only if it preserves the cor-

## A Just-in-Time Compiler

*SooMook Moon, Seoul National University*
*Kemal Ebcioğlu, IBM Research*

Programs written in Java are typically translated into bytecode, so that binary distributions of Java applications can be ported to any platform for which a Java virtual machine (VM) is available. One approach to executing Java bytecode involves compiling a set of bytecode instructions to native code just in time to be executed. Compared to using an interpreter, just-in-time (JIT) compiling can drastically improve performance.

Latte (http://latte.snu.ac.kr/) is a VM with a JIT compiler that does traditional and object-oriented optimizations. Latte uses a two-pass algorithm on each extended basic block. In the first pass, a backward sweep constructs a preferred mapping of symbolic registers to actual registers, and in the second pass, a forward sweep constructs the actual mapping. This algorithm produces good results with low overhead.

The JIT compiler is not the only thing that determines the performance of a Java VM. Thread synchronization, exception handling, and garbage collection are also important. If these functions are slow, a JIT compiler can only do so much to improve performance. To wit, Latte uses a lightweight monitor for thread synchronization, on-demand translation of exception handlers, pointer incrementing memory allocation, and a fast, partially conservative mark-and-sweep garbage collector. Consequently, the performance of the Latte VM is competitive with that of commercial Java VMs, such as Sun's Hotspot and JDK 1.2.

Latte is a research prototype and has been used to test several optimizations, including fast register allocation, reduction of virtual call overheads, instruction scheduling, adaptive compilation, and new memory-allocation and garbage-collection algorithms. Latte has a liberal BSD-like license. We hope others will use it as a research framework.

rectness of the translation. Correctness and performance are often competing issues.

## LEGACY CHALLENGES

Since all machines—legacy and new—are Turing machines, any computation done on one can be emulated on another. Binary translation aspires to do more than emulate the legacy architecture efficiently. Binary translation seeks to emulate legacy architecture so efficiently that code runs at least as fast on the new architecture as on the legacy machine.

Attaining this goal requires careful design in several areas. Executing legacy architectures efficiently is a difficult task since architectures are never fixed. The sidebar "Translating MMX" describes one such architecture change.

### Collaborative Profiling

*Youfeng Wu, Intel*

To accelerate performance, profile information should be aggregated and ready for use in optimization. The traditional approach to profiling is to sample the program counter periodically and save it into a file. Even though these common sampling-based techniques have low overhead, they often require postprocessing to derive the profile information. One approach to avoiding this problem combines two collaborative techniques: compiler analysis that inserts profiling instructions and hardware that asynchronously executes profiling operations in free program slots.[1]

The compiler first analyzes the user program to determine the minimal set of blocks that need profiling. Then it allocates profile counter space for each function and inserts an `iniprof` instruction in the function entry block to tell the hardware the starting address of the profile counter space. This compiler inserts a profile ID instruction in each profiled block to pass profiling requests to the hardware. The hardware derives the profile counter address from the `iniprof` and profile ID instructions and generates profile update operations. The compiler executes these operations when free slots exist.

Since most profiled blocks contain a branch instruction, this approach can eliminate most profile ID instructions by encoding the ID into eight bits of a branch instruction. For large functions that require more profile IDs than the eight-bit ID field can hold, you can apply a graph partitioning algorithm to partition the control flow graph into single-entry regions. The ID values in each region can be encoded into the eight-bit ID field.

This approach lets programs with profiling run almost as fast as without profiling. Our group observed less than 0.6 percent overhead on one set of benchmarks.

Reference
1. Y. Wu, Y. Lee, and H. Wang, "An Efficient Software-Hardware Collaborative Profiling Technique for Wide-Issue Processors," *Proc. 1999 Workshop on Binary Translation*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 5.1-5.10.

## Memory

The number of registers in the new machine should be at least equal to the number in the legacy machine. If the new machine has fewer registers, then some legacy register values must be kept in memory with costly loads and stores used to access them.

The system states, stored in special-purpose registers, must also be maintained in the new architecture. For example, if the legacy architecture has special seg-

### Native Binary Acceleration

*Vasanth Bala, HP Labs*
*Evelyn Duesterwald, HP Labs*
*Sanjeev Banerjia, InCert*

Dynamo, a prototype optimizer developed at Hewlett-Packard (http://www.hpl.hp.com/cambridge/), transparently accelerates native program binaries. Dynamo operates entirely at runtime by dynamically generating optimized native retranslations of the running program's hot spots.

Contrary to intuition, Dynamo demonstrates that you can use a piece of software to improve the performance of a native, statically optimized program binary during execution. For example, the performance of many SPECint95 binaries compiled with moderate optimization running under Dynamo compares favorably with the performance of their statically optimized version running without Dynamo.

Dynamo operates transparently in the sense that it does not depend on any special annotations, compiler-generated hints, binary instrumentation, or OS and hardware support. Dynamo uses lightweight profiling techniques to identify hot traces at runtime, monitoring program behavior until the target of a backward-taken branch becomes hot; at that point, Dynamo selects the very next sequence of interpreted instructions as a hot trace. The selected traces form a single-entry-multi-exit region of code optimized using low-overhead optimization techniques and emitted into a software code cache.

Traces may cross statically and dynamically linked procedure boundaries, potentially exposing optimization opportunities not available to a static compiler. Over time, as the program's working sets materialize in the software cache, execution takes place almost entirely inside the optimized code cache, thus giving a performance boost.

Operating on HP-UX, Dynamo has a code size of less than 265 Kbytes. Its compact memory footprint makes Dynamo deployable in a wide range of platforms.

mentation registers, they should be mirrored by the target architecture. You must also deal with flag registers that maintain condition codes. One simple approach is to have the new architecture set them in the same way they are set in the legacy machine. Alternatively, a translator may be able to use a combination of redundant flag elimination plus smart flag calculation.[2]

Memory-mapped I/O presents a problem particular to whole system translators. References to I/O locations can have side effects such as injecting a packet into a network or turning on an alarm, and must be done in program order and without caching.

### Architecture

Many architectures contain instructions that must be executed atomically with respect to memory; this means that a second processor cannot access memory while an instruction is executed. Replicating these semantics on a different architecture is often difficult.

Noninterruptability causes problems similar to atomicity. Some architectures, such as the IBM S/390, have complex instructions that take many cycles to execute and must execute completely or not at all. The new architecture can prerun translations of such instructions without side effects to guarantee the real run will work.

The noninterruptability problem is one instance of the more general problem of dealing with precise exceptions for the legacy machine. For example, if a particular memory load causes an exception, the legacy exception handler will expect all instructions prior to that load to be completed but none of the instructions following the load to be completed. Precise exceptions are harder still if translated code is reordered.

### Code discovery

Precise discovery of the legacy code also poses a problem, especially for static translation. Given an executable file, it is not always clear what is code and what is data. Although the general problem of code discovery cannot be fixed, various tools have used value tracking and other methods to solve the problem successfully in many practical situations.

A problem related to finding legacy code is self-referential legacy code—code that looks at itself, for example, to perform a checksum. Because of the nature of the code, a copy of the legacy program counter must be maintained by the new machine and used whenever the legacy machine references the counter for anything other than an instruction fetch.

Self-modifying code presents problems similar to finding legacy code and self-referential legacy code. Handling self-modifying code is not possible with a purely static translator, and is not always easy even with a dynamic translator: The new architecture must provide some means to detect modification of legacy code so that translations corresponding to the modified code can be invalidated.

### TRANSPARENT INTEGRATION

Ultimately, binary translation seeks to provide the illusion of transparency: Code can run on platform A exactly as it would on platform B. Most aspects of transparency described thus far relate to instruction-set conversion. But there are several other aspects.

## Translating MMX

*Paul J. Drongowski, Compaq*

MMX instructions in the x86 architecture aid multimedia code by performing multiple parallel operations on four 8-bit or two 16-bit operands. Compaq's FX!32 lets you run 32-bit x86 Windows applications on Alpha. To support MMX in FX!32,[1] we developed an Alpha code template for each MMX operation. Actually, we developed two sets of such templates—one for the 21164 and one for the 21264 Alpha processor. Since the 21264 has its own multimedia extensions for value saturation and packing, some templates were trivial projects. Other MMX operations, like the 64-bit logical operations, are part of the Alpha ISA and hence also have templates that are easily created for the 21164.

One big decision entailed determining the physical location to which x86 MMX registers would be mapped on Alpha. Although all Alphas have 32 64-bit integer and 32 64-bit floating-point registers, the 21264 has instructions to move values between integer and floating-point registers, while the 21164 does not. The 21164 must read and write values to memory in order to transfer them.

We considered three choices: storing the MMX values in floating-point registers, in integer registers, or in memory. Since values must be moved to integer registers to execute, the first option is bad for the 21164. The second option leaves fewer integer registers for all translated code, even code with no MMX instructions. Storing MMX values in memory leads to higher load-and-store traffic and more cache misses.

We decided to store MMX values in floating-point registers since 21264-based machines were shipping, and future memory optimizations could reduce the penalty on the 21164. Further, intermediate results could be kept in integer registers by postponing updates to MMX registers until necessary.
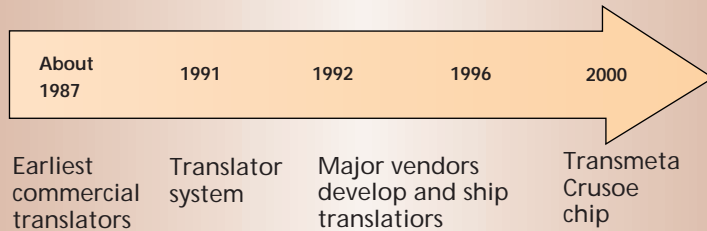
To get a fast first implementation, we wrote simple subroutines, a style already in use for high-complexity x86 string and floating-point operations. Unfortunately, the overhead proved quite high. Execution took longer than non-MMX execution. To remedy this, we inlined MMX instruction translations instead of calling subroutines. The resulting performance on a 500-MHz 21264 was superb, beating a 266-MHz Pentium II (3.77 versus 6.24 seconds). Performance on the 21164 lagged behind non-MMX execution. Measurement revealed the lost cycles were due to store and load penalties.

**Reference**

1. R.J. Hookway and M.A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," *Digital Technical J.*, Vol. 9, No. 1, 1997, pp. 3-12.

A user may wish to run an application compiled for one OS on a different OS. For example, you might want to run a Windows productivity application on a Unix workstation. You might want to move an application between different flavors of a single OS—a Win32 application to a 64-bit Windows environment, for example.

Translated applications can be cached in persistent storage but must preserve the normal semantics of an executable file. For example, any initializations done in invoking the executable must still be performed. Likewise, if the executable file is updated, the translation must be as well.

A legacy platform may provide instruction semantics that must be emulated on a newer platform. For example, direct I/O instructions cannot be executed directly on the hardware in most modern OSs. Solving this issue typically requires emulating legacy hardware devices.

### Complete OS emulation

OS emulation can be dealt with in several ways. One way to bypass most complications involves emulating the entire legacy OS. This approach results in a structure similar to a virtual machine.

The concept of a virtual machine dates back to the early 1960s[3] and has recently come back into vogue.[4] A classical virtual machine lets code run freely as long as no privileged instruction executes.

For a privileged instruction, a special code sequence emulates the intended operation, possibly using the primitives of an underlying OS. When integrated with binary translation, most code in a legacy OS would be translated normally, with special code sequences executed for privileged instructions. As modern virtual machines stabilize, we expect to see integrated solutions consisting of a binary translator interfaced to a virtual machine.

The approach of complete OS emulation is practical if, as in the Daisy emulator,[5] the new platform's whole purpose is to emulate a legacy platform using a very fast CPU. This goal is not always practical since the cost of emulating the entire OS may be prohibitive for some systems.

### Application-level translation

A more common solution involves a port of the OS to the new architecture, with binary translation employed only at the application level. The new OS would run both native and translated legacy applications.

When the OS identifies a legacy executable, it launches the translator. Afterward, most work can be done at the application level. Note that when you use static translation, the OS can still provide transparency by diverting execution from the legacy executable into the pretranslated native executable.

When you migrate an application between similar OSs—say, from one Unix to another—several issues must be resolved at the application-OS interface, including

- calling conventions, because one machine may pass parameters in registers and another may pass them on the stack;
- memory mapping, because the OS may have trouble locating the application's stack; and
- memory alignment, where different architectures have different requirements.

These three problems multiply when a translated application needs to communicate with a native application directly through shared memory or through another interprogram communication mechanism.

When the application needs to be migrated to a different platform, you can use a jacket layer to transform OS calls from the semantics of the legacy OS to the new one.

We outlined several innovations in binary translation, but we believe the field still has improvements to make. We need to

- improve translation from virtual instruction sets like Java virtual machines,
- provide full system translation that more efficiently translates the entire operating system,
- use more sophisticated profiling techniques to

adapt to behavior changes quickly, and

- support noninterruptibility, atomicity, and multiple legacy architectures in one target.

We also need to provide additional information to the translator beyond that given by object-code semantics. Java class files have moved in this direction already, mostly to facilitate safety and security analysis. We need to develop new compilation techniques that benefit directly from dynamic code generation. ✳

References

1. J. Kalamatianos et al., "Analysis of Temporal-Based Program Behavior for Improved Instruction Cache Performance," *IEEE Trans. Computers*, Feb. 1999, pp. 168-175.
2. M. Srinivasan, "Method and Apparatus for Emulating Status Flag," US Patent 5774694, June 30, 1998.
3. R. Goldberg, "Survey of Virtual Machine Research," *Computer*, July 1974, pp. 34-45.
4. M. Rosenblum et al., "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. Modeling and Simulation*, Jan. 1997, pp. 78-103.
5. K. Ebcioğlu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proc. ISCA 24*, ACM Press, New York, 1997, pp. 26-37.

*Erik R. Altman is in the high-performance VLSI architecture group at IBM T.J. Watson Research. He received a PhD in computer science from McGill University. Contact Altman at erik@watson.ibm.com.*

*David Kaeli is an associate professor at Northeastern University. He is also the director of the Northeastern University Computer Architecture Research Laboratory (NUCAR). He received a PhD in electrical engineering from Rutgers University. Contact Kaeli at kaeli@ece.neu.edu.*

*Yaron Sheffer is a researcher at Radguard, a network security company. Earlier he headed a binary translation project at Intel. He received a BA in computer science from the Technion and an MSc from the Hebrew University in Jerusalem. Contact Sheffer at yaron.sheffer@radguard.com.*