Binary translation : Classification of emulators

Arjan Tijms
Leiden Institute for Advanced Computer Science
University Leiden
atijms@liacs.nl

Abstract
This paper introduces the concepts of emulation in general. Emulation is being used today across wildly different platforms and at various levels of a system. Due to this diversity a wide range of different techniques have been used to achieve the goal of emulation. In this work we approach the field by classification of the different kinds of emulation and identifying the methods that are used to overcome the most general problems. The focus will be on emulation and classification, whereas the other forms of binary translation will be shortly described for the sake of contrast and methods will mainly be used as illustration. Exact details of how things work can be found in the works listed in the references.

1 Introduction

Today's rapid innovations in both software and hardware systems lead to an ever-increasing amount of available technologies. However, these technologies cannot simple be used together in an arbitrary way. We usually see that adapting these new technologies means abandoning an existing software or hardware base.

This problem has been attacked in various ways. In past years the trend in software architecture has been towards modular and component based designs. System specifics are isolated in order to facilitate easy porting and operating systems are built with retargetable APIs and open device drivers. Nevertheless porting remains a troublesome and far from transparent undertaking. Furthermore, source code is not always available, especially not when off-the-shelve objects have been used which came without any source code.

There is also the related issue that at the time of porting the target architecture may not have sufficient resources to run the legacy app at full speed. So more often than not developers choose to drop certain features of a ported application or do not port at all. When after some time the target platform does get enough resources or certain components available for it, the legacy app is only rarely re-ported or updated. With a series of techniques called binary translation we can circumvent these problems by letting legacy software run transparently in another environment than the one for which it was originally intended by using the original binaries. We shall first give an overview of the various options to achieve this and shall than examine its details further.

We can divide the world of binary translation in the following two main classes:

1) Emulation
2) Translation

These can be subdivided and combined in various number of ways as described below.

2 Emulation

Emulation is the art of presenting an environment to a binary, which behaves just like its original target environment.

This class of binary translation is mainly concerned with the field of computer architecture. It involves the modeling of hardware components, like a CPU, system chips (DMA, timers, interrupt controller, bus controller etc), and various subsystems (audio, video, I/O) in software. The combination of these software components is often referred to as a virtual machine. We either say the virtual machine is running- or the machine which hosts the VM is interpreting- the binary, depending on our point of view.

Emulation now works by reading each instruction from the binary as the original machine would do, execute it in some way, and set the virtual machine to a similar state as the genuine machine would be in after executing this instruction. Note the use of 'similar' state instead of 'exact' state. Depending on the apps we target, we can choose to ignore a part of the machine state which is not being used by the binary. This is what we call the accuracy of the emulation.

2.1 *accuracy*

We can differentiate between 5 global levels of accuracy:

1) Datapath accuracy
2) Cycle accuracy
3) Instruction level accuracy
4) Basic block accuracy
5) (Very/ultra) high level emulation accuracy (usually used by its acronym HLE, and prefixed with Very or Ultra depending to which extreme its being used)

Datapath accuracy
The first level of accuracy involves modeling the internal structure of components which can not be seen from the outside. This level is mostly used by hardware designers to test new architectures and is mostly referred to as simulation. For binary translation such accuracy is rarely required, perhaps only in the case when emulating hard real time systems with extreme synchronization problems (e.g. when the original hardware has such strict and fixed execution times that software could depend on race conditions). Especially on older embedded systems with limited resources, programmers have gone to extremes to squeeze the last bit of performance out of these chips. Apps using such 'tricks' are usually very hard to emulate.

Cycle accuracy
The second level involves modeling the system (or some of its parts) in such a way that instructions complete in the same relative amount of time compared to each other and to other components in the system. This level is often needed for accurate emulation although usually not strictly required. Depending on the machine targeted, emulators without cycle accuracy can run most of the applications for it with some having only slight deviations from the original (e.g. an icon which appears misaligned on screen by 1 pixel because the background has already scrolled) and only a few not working at all or being unusable (e.g. when the screen background is drawn before the foreground). Often we see that an emulation writer handles these last mentioned cases by providing individual patches for the original code if the incompatibility concerns only a few known apps.

Instruction level accuracy
The third level models everything a genuine instruction can see and touch but without the strict timing constrains. E.g., a complicated divide instruction could possibly complete just as fast as a simple add in terms of the virtual machine timing.

Basic block accuracy
The fourth level is not really used by a pure emulator, but rather in one used in combination with dynamic translation (described below). The idea is to replace a basic block with one in native

code having the same outputs for a given input. This requires some profiling/analysis of a specific binary and thus cannot be a static part of the emulator.

HLE accuracy
The fifth level works like the fourth, but instead of replacing basic blocks it replaces larger blocks that have some specific semantics. These blocks contain high-level functionality provided by the original system, which could be anything, but in practice it's usually restricted to audio or video output. The criteria for this to function well could be that the (local) state change caused by such HL-blocks is not referenced in any following code. E.g., a series of register writes to a graphics processor could be recognized as 'setting up and displaying polygon'. Such a block could be replaced with a native method for doing this, eliminating the need to emulate the legacy GPU. Contrary to finding basic blocks, these higher-level blocks can't in general be discovered automatically and transparently. One method used is tracing entire routines generated by development tools used for the legacy platform and trying to recognize these in the legacy binary. Another method is inspecting of code by a human programmer who can discover certain blocks by understanding its 'meaning'. Such a block is than marked and replaced at runtime by a function call providing high-level emulation. This scheme can be generalized by hardcoding known HL blocks/patterns in the emulator.

At runtime known blocks can be recognized in new/untested apps and -hopefully- applied the right HLE to. In addition locations for specific blocks for specific apps can be read from file. These files can also include small patches for certain apps. This way new apps can be supported with only minimal efforts and without rewriting the emulator.

Naturally a higher accuracy makes for a more compatible emulation but it also requires a lot more processing power. Datapath accuracy for an entire system can easily demand several thousand times more processing power when the performance of the emulation must match the legacy's one. When recent architectures are being emulated HLE may be the only way to achieve reasonable speeds. However HLE breaks the advantages of emulation by requiring a porting like process for each app.

As an example of emulators that require enormous amounts of performance, we look at Arcade and console entertainment system emulators. This category of emulators represents essentially hard real time complete systems. Every application designed for such a system is designed to run at a fixed system at an exact speed. Furthermore these systems often use very complicated designs with a large number of components co-operating very tightly. While the design of these systems is essentially hard real time, in emulating them we can resort to soft real time. We can choose to simply drop some output (i.e. skip some frames) or slightly alter the speed of a system as a whole, thereby leaving the relative speeds of the individual components the same.

It's interesting to note that absolute performance of this kind of systems is not always a measure for the amount of performance needed to emulate them. E.g. a SNES system running at 3Mhz, utilizing 256 colors and a fairly limited amount of sprites & objects requires about a 550Mhz Pentium to run a full speed emulation. In contrast, a much more powerful NeoGeo utilizing a 10Mhz 68000, 4096 colors and much larger and more sprites, requires 'only' about a 350Mhz Pentium to run a full speed emulation. This is due to the SNES being build out of a lot of cheap components with tight co-operation in contrast to the NeoGeo's few expensive and powerful ones.

2.2 *Completeness*

When providing an environment in which a binary can run we don't always have to emulate all of its parts. Either software or hardware parts may already exists in the target environment. If they don't, it's often quite feasible to (re)create a few parts specifically for the new machine instead of taking the emulation path. This usually does require great efforts but just as the emulator itself, it only has to be done once for a multitude of applications. Such a component can then be used for

running legacy code only but in some situations can be used by native applications too.

We shall first look at a classification of above-mentioned components. There are 4 global levels which we recognize:

1) Completely hardware based
2) Mainly hardware based
3) Mainly software based
4) Completely software based

Completely hardware based
In this category we place hardware solutions which aren't accompanied by any software running on a host's processing unit. However internally some form of software can be still be used. If we mainly look at complete systems, we often see these aren't emulators at all but rather a system-on-a-card or recently system-on-a-chip. The only things these solutions share with their hosts are usually things like casing and power source. If we look at a single component, we do see some interesting solutions:

- A special processor mode for emulation of an older generation CPU. For example the so-called real mode in Intel x86 class processors or the on-chip x86 code emulator in AMD's K5 processor. Disadvantages are the fact that processor mode switching is usually slow which hurts performance in a multitasking system. Implementation is also costly in terms of die space and design difficulties.

- Hardware cracking: The core of a processor works with a different instruction set than the one that is used to program it by outside binaries. This is usually the case when a design evolves from classic CISC to a modern RISC architecture. Internally the complex instruction is broken up in a sequence of simpler RISC-like instructions (often called micro-ops) that can be pipelined and individually scheduled out of order.

- Microcode emulation: A processor can be (partly) micro-programmed in order to support other instructions than those from its own ISA. Contrary to hardware cracking, this exposes both (or all) ISAs to the outside, which means programs can theoretically use instructions from each ISA. Often we see that the main ISA is hardwired. Due to the impossibility to hardwire a large set of instructions, additional ones are than microcoded.

Mainly hardware based
Here we find solutions that consist mostly out of hardware but require a thin, yet essential software layer to work. If we look at the complete systems again, we find near system-on-a-card solutions. E.g., a board consisting out of a CPU, memory, system ROM, system chips, and some I/O. A common set of I/O however is provided through emulation by the host. This usually includes keyboard/mouse services, disc storage and display capabilities. Looking at individual components we see:

- Hardware virtual machine: The concept of a virtual machine can be directly supported in hardware. E.g. the so-called virtual 8086 mode in Intel 386+ class processors. The idea is to provide a set of virtual registers, virtual memory, virtual interrupts and behavior (e.g. with respect to address calculation) just like the original machine but in the protected context of the new one. The difference with simply switching to a legacy processor mode is that after such a switch the complete machine (or at least the addressable and visible part if the address range of the legacy machine is lower than the range of the new machine) would be open to the application in that mode. This could undermine any protection mechanisms the host operating system may have in place. Hardware virtual machines must be paired with a virtual monitor, a small piece of software that manages access to shared system resources. Each virtual machine can have its own legacy OS,

but if the hardware doesn't provide virtual interrupts its more convenient to provide an emulated OS running as native task on the real processor.

Mainly software based
This class of emulators consists mainly out of software but can make use of some available hardware in the target system which happens to be the same as the source systems. For example, in the 80ties a lot of systems used Motorola 680x0 processors (e.g. Sun, Apple, Commodore, Atari). Emulators between them could easily achieve a high level of performance, since 'only' the periphery needed to be emulated. As a consequence, the machine emulated would not always be an existing or intended one since it depended on the type of host CPU. E.g., an emulator for a Macintosh providing the periphery of a Mac classic would, when running on a 68040 CPU, create a 68040 Mac classic. This is a machine that never existed in reality. Taking this one step further, the only way to use a 68060 CPU on a Mac was through emulating it on another machine, since Apple moved to the PowerPC before the 68060 came out.
More recently we have seen that certain PC emulators on the Macintosh platform (e.g. Virtual PC up to version 3.x) could use specific video cards (e.g. 3dFX Voodoo2) directly since both platforms use the exact same technology.

We sometimes see that some performance/compatibility critical hardware component is specifically created for use with an emulator. For example, an old Macintosh emulator Emplant came with a card including some simple timer hardware. Despite its simplicity, emulating those in software would have been very expensive in terms of performance. As a more recent example, Sony has included an identical copy of the PSX-1 34Mhz MIPS CPU in its new PSX-2 platform. So as not to waste silicon, this former CPU serves as the PSX2's I/O processor when running new games, switching to the role of central processor to run old games.

Completely software based
Despite being the most power demanding, emulators that don't require any specific hardware remain the most popular group. This is mainly for two reasons:

1) Adaptability
2) Portability

Adaptability means an emulated component can be easily reused among emulators for different systems. For example, an open-source implementation of a 68000 emulator can be easily adapted to power systems with various precise clock speeds, e.g. 7.16 MHz for an Amiga, 12.00 MHz for a NeoGeo etc. Moreover, such implementations can be easily adapted to emulate a different CPU from the same series, e.g. a 68020 used by the Taito F3 system. Finally, duplication of such components is a trivial matter of multiple instantiations.

Portability needs little explanation. It simply means an emulator can be carried over to a new platform once the original host platform becomes legacy itself. Note that this doesn't mean emulators are inherently easy to port. Porting an emulator is as difficult a job as is porting any application.

2.3 *Operating system influence*

A very special environmental software component is the operating system. Most modern and some older operating systems are written with device independence and retargetability in mind. This means that any application that solely uses the OS services and APIs instead of depending on any specific hardware runs transparently on different machine configurations providing suitable device drivers exist. Most modern operating systems even forbid direct hardware use due to stability and protection issues. This situation can be used to the advantage of emulators in various ways.

First of all, we can sometimes port an OS to our new architecture. The model example here is the Unix operating system, which has been ported to a lot of systems, and to some extend Windows NT which was portable in versions prior to 5.0 (also known as Win 2000). Running the OS natively instead of in the virtual machine relieves us from an enormous emulation burden and offers various advantages:

- All hardware devices required can be used directly. This not only relieves us from emulating them but also offers the choice of using any supported device for specific classes (e.g. audio/video hardware). Emulators in general only emulate a single common device of each type.
- All of the system calls an application makes will be run by native code. Especially interactive applications, which spend a lot of time executing system code, will have a large gain here.
- A legacy app can communicate transparently with a native app running on the same OS. If the legacy was running under its own (emulated) OS, only virtual network communication is usually possible

Of course we still need to emulate the processor for which the binary was coded. Since no OS has standard defined mechanisms for CPU abstraction we still need to provide a virtual machine, although it would include only a CPU. If we want true transparency, the emulated app must appear to the system as a local process. This is far from trivial. It involves building the virtual machine as a kind of shared resource(s), which are implicitly used by the legacy app. This preserves its identity to the system, but introduces some new problems (see below).

2.3.1 *System calls*

Even though appearing to run as a native process our legacy app is nonetheless still executing inside a virtual machine. The consequence of this is that system operations and structures cannot be mapped into the address space of the application as normally would happen. Instead so-called jackets are used which provide an interface to the systems native-processor based OS calls. This works by first inserting references to jacket code at places where OS modules (i.e. DLLs) are conceptually imported. This is often called the 'jacket layer'. Depending on the OS this can be for example a file import table (Windows) or common system-call gateway page (HP-UX). At this place a jump is made out of the virtual machine, for example by starting the jacket code with an illegal opcode which traps to an exception handler in the virtual machine control context, i.e. the runtime/environment emulation module. At this point we can either 'directly' invoke the native system call or provide some special handling first. Either way several issues have to be resolved with respect to parameter passing and return value receiving:

- Calling conventions: values may be passed on a stack or in registers. This is usually not a part of the general OS specification, but rather part of the OS specifications for a particular machine.

- Memory mapping: If values are passed on a stack in both the source and target machine, there might be differences regarding location of the stack itself and ordering of values here in. E.g. parameter 1 may be on the bottom of the stack or at the top and bottom may be either at the lowest address (up counting) or the highest (down counting).

- Register mapping: If values are passed in registers, some scheme might be needed to map between source and destination registers. E.g. the source machine might place parameters sequentially in general purpose registers, whereas the target machine expects specific parameters in specific registers, like addresses in A0~A8 and data in D0~D8 for example.

- Memory alignment: The source architecture may have different alignment requirements than the target architecture.

- Endian conversion: Machines can be either little- or big endian (some, like the PowerPC, can be both). If they differ, a series of byte swaps are necessary.

- Data conversion: In some rare occurrences, like when a non-retargetable OS is used, the OS might expect data passed to have a certain format. E.g. planner to chunky display data conversion. In some older systems pixels are organized in bit planes, meaning the first bit of every pixel is placed in one field, the second bit in another etc. i.e. the first byte of field 0 contains all first bits of pixels 0~7. Other systems have their bits sequentially in one field i.e. the first byte of the (only) field contains the first 8 bits of pixel 0.

As mentioned, some system calls require special handling before the native system calls are made. For example, when a thread requests the suspension of another thread this cannot simply be passed as-is. The thread does not know whether itself or the other thread is executing in the virtual machine and so might have implicit possession of shared VM resources that the thread has no idea about. Suspending such a thread could cause a deadlock, so the VM control context must first acquire all shared resources before sending the native suspension request.

2.3.2 *Signals & Interrupts*

Special care must also be taken when handling signals and interrupts. Because of the interpretive nature of an emulator, signals associated with a particular instruction can usually be delivered precisely. However, due to performance reasons we might not always construct a correct signal context after interpreting each instruction. Whenever the OS sends a signal to our emulated app, we first need to construct such a context, and only thereafter can we pass the signal on to our application. In the case data is send with a signal, we might need to apply the same conversions as are needed for system calls.

2.3.3 *Partial OS porting*

In many circumstances however we might not be able to port an operating system. Porting an operating system can be an enormous effort. After the initial port has been realized we face a number of problems. Ideally a lot of device drivers are created for the ported OS, addressing hardware found on the new machine. However, vendors of such hardware may have only experience in writing drivers for the original OS of that machine or may simple not want to support the new OS. Furthermore, in order to gain any momentum, the ported OS should have a fair amount of native applications created for it. Maintaining, marketing and supporting an entire OS for the sole purpose of emulation is sadly not always economical feasible. Examples include the discontinuance of Windows NT for the Alpha platform due to lack of sales (the majority of Alpha machines run a Unix variant) and lack of native applications (few came out at all, and the ones which did where always a version behind their x86 counterparts).

So instead of porting the entire OS, we can opt for a ‘normal’ machine emulation which runs the original legacy OS and provide native implementations for some of its parts. We can do this by:

- OS retargetability: Target our native component directly through a custom driver

- Component porting: Direct a call to certain system functions to a ported component running natively. This can happen in much the same way as we direct system calls in the case of a complete port. The only difference is that now only some calls are redirected instead of all calls.

For example, Insignia solutions' Softwindows (recently taken over by FWB), which emulates a Pentium class PC on a Macintosh computer, provides various drivers and partial ports for Windows 95/98. Among others it implements its own TCP/IP stack for Windows running natively on the Mac hardware (by porting winsock.dll), and provides a sound driver which makes the Mac sound hardware directly useable under windows without any emulation, while the display driver runs partially native. Some non-driver based functionality has been ported, like the floppy format routines.

The advantage of this approach is a speedup compared to complete system emulation. The downside however is that compatibility suffers. Adding and tapping in an existent OS could cause some problems, especially when the emulator developer does not have access to OS source. Furthermore use is limited to supported operating systems, either new versions or updates/patches of the legacy OS could break the emulator.

2.4 *Outside environment*

One point even a complete system emulator does not address is the characteristics of the environment outside of the digital system. Purists argue that these should be taken into account too when developing a system emulator. Although this may not be an exact science as seen from the viewpoint of the binary, it does has much ground in common with the field of physical simulation. The outside environment mostly concerns audio and visual signals and how the user of the system perceives them. For example, older systems were connected to monitors with very high dot pitches and coarse phosphor elements. As a result, images appeared with a certain blur on screen. Artists designing graphics on such systems took this natural blur effect into account while painting.

When these systems were later emulated on machines utilizing low dot pitch, high resolution monitors these same graphics appeared enormously pixellated, even to such a degree that the original patterns of objects were no longer recognizable. For example, an old imaging system used for creating digital effects for television commercials showing what almost looks like a random noise display in an emulator. Only by applying a blur filter (e.g. Adobe Photoshop Gaussian Blur, ~0.8 pixel radius) the clear patterns of a grassland appeared with all its subtle highlights and shadows.

The same kind of affect can be witnessed when emulating old systems used for creating music. Although the emulation of the digital system may be cycle accurate or even better, the music generated by it may sound not at all like the original machine. In this particular case the emulator would be useless when its original purpose was to recreate the original sound.

3 Translation

Translation is the art of transforming a binary in such a way that the semantics of the result equal those of the original exactly and precise.

This class of binary translation is mainly concerned with the field of formal languages. It involves parsing the bits that make up the binary executable on a source machine into their linguistic elements for which the semantics are known. Those elements can than be compiled and assembled to a new binary executable on a target machine. The process takes place offline, i.e. without running the executable, and is therefore named static translation. This in contrast to dynamic translation that is described below.

Another term for dynamic translation that has come into fashion is dynamic recompilation (mostly used by its acronym DynaRec). This hints that translation is some form of (re)compilation, and in fact it is. A static translator like University of Queensland's UQBT is a good illustration of this. In

general a translator takes a number of (conceptual) highly sequential steps with each step coming closer to the end result. If those steps are really done sequentially we call them passes. Often however we see that several passes are either combined or only taken implicitly. For the sake of clarity we will assume they are all taken individually.

3.1 *passes*

We can differentiate between 7 global passes a static translator can take:

1) Bit grouping
2) Disassembling
3) Intermediate code generation
4) De-compilation
5) Compilation
6) Assembling
7) Bit storing

Bit grouping (code discovery)
Using knowledge about the format of an executable and some heuristics, groups of bits are retrieved from the executable which make up the programs code section(s). Precise code discovery is generally an unsolvable problem, but various techniques have been very successful in practical situations.

Disassembling
The found bits are disassembled/decoded are transformed to instruction-operand(s) pairs. For this pass a description is needed about the binary representation and possible operands of each instruction.

Intermediate code generation
The assembly instructions of the previous pass are transformed to a machine independent representation. At this level the language is still assembly-like (e.g. three address code) but abstracting from any machine specifics. This is sometimes called the semantic specification

De-compilation
This is a special optional pass. Here we transform the intermediate code from the previous pass into high-level language statements (like C or Pascal code etc). This is the highest level that we reach. From hereon we can use traditional and well-understood compiler technology to carry out the remaining steps of the transformation. If we omit this pass, we have to generate the target code (at least the assembly code) ourselves from the intermediate representation.

3.2 *Completeness*

Usually translators only work between languages of a single kind of component that in the majority of the cases is a general purpose processor and sometimes a media processor. Languages of these components are well defined enough to facilitate automatic translation. However the sub-class of these languages using so called self-modifying or self-referential cannot be translated. Especially self-referential code is often used when calculation checksums.

Semantics of other programmable devices (system chips, video hardware etc) are not yet tightly enough defined (if at all!) to make an exact and precise translation possible.

Since we can't yet translate between two arbitrary kinds of components, there is no ground at all to reason about a possible static complete machine translation. This would possible involve defining the semantics and a language for a system itself in terms of its subcomponents. Although some work has been done at this level with studying HLE (see above), this is not yet a reachable goal.

3.3 *Operating system influences*

A translated application usually has to work under the same operating system as the source binary did. Static translation between different operating systems is generally not possible.

3.3.1 *System calls*

Static translation can address some common problems regarding different conventions for multi-platform operating systems on different machines. This include parameter passing issues like calling conventions, memory mapping (distribution of the stack frame)

3.3.2 *Signals & Interrupts*

Due to optimization and the fact that there's not always a one to one correspondence between a source instruction and its equivalent piece of code in the target binary, static translated applications can't support precise exceptions.

3.4 Combining

As we have seen there are many problems with static translation that cannot be easily overcome if at all. Therefore a pure static translator does not really exist but is always combined with some form of emulation, either to support handling undiscovered code, address calculation for data sections (which need the original program counter), precise exceptions etc.

We can distinguish between two such combinations:

1) Mainly static translation based
2) Mainly emulation based

Translators that are mainly static based work like a pure static translator but include some interpretation hooks for the problem areas. They have the advantage of being able to first translate a binary offline, where the optimization process can theoretically run for weeks if necessary, trying to discover every last piece of parallelism.

Translators that are mainly emulator based are called dynamic translators and are the topic of most recent research. The basic process is the same for all variants:

-Start with interpreting the code.
-Monitor the execution to find out which code executes frequently.
-Schedule this code for 'simple' translation if possible.
-Possibly optimize if the optimization gives enough gain relative to the cost of optimizing.
-At future references execute the translated code.
-Whenever an exception occurs in translated code, set the correct state by executing the untranslated block.

Note that some dynamic translators can in fact be classified as using self-modifying code, since they execute their own generated code. This is especially true when seen from the viewpoint of a translator being part of an application's resources. SUN recognizes this fact with its [Magic] processor that has special support for self-modifying code generated by Java JIT compilers (which are a type of dynamic translator).

To sum up, the hybrid approach offers the best of both worlds, the compatibility of emulators combined with the speed of translators.

4 Alternative uses

We would like to mention a few alternative uses to binary translation. Binary translation or some of its techniques is not just used for running legacy apps on new architectures. It's also used in the following ways:

-Native virtual machines: A native virtual machine hardly emulates anything except for privileged instructions. Its use is to create a number of independent environments hosted by one controller. These native VMs can for example be used to run a number of different operating systems at the same time at the same machine. An example is VMWare. It 'emulates' a complete PC from the viewpoint of an OS executing inside it, but in reality a lot of resources are directly mapped to actual resources inside a PC.

-Native binary acceleration: A binary translator that translates a running program's hotspot in optimized versions, just like dynamic translator but in the same language. An example is HP's Dynamo. Essentially these are run time optimizers. Collected profiles can sometimes be used by a compiler to recompile the binary, but more often they strictly work dynamically i.e. optimizing for the current behavior.

-Software instruction decoding: Decoding of instructions just as for example x86 processor do when cracking complex CISC instruction into RISC like micro ops. An example is the code morphing software of Transmeta. This works more or less like any dynamic translator. The difference is that translating is not just an option here but the entire strategy. The translator runs below any operating system and is the only application running natively on a Crusoe platform.

-Java JIT compilers: Essentially a dynamic translator, a Java just-in-time compiler interprets and translates instructions from one machine to another. The only difference is that the source instruction were never those of a legacy machine but of a 'non existing' (or only in software existing) machine which is emulated by all others. Usually this machine is only compiled to from the Java language. I.e. you normally don't compile a C++ program to this machine. The idea has generated some spin-offs, like the small JavaOS and processors with the Java bytecode as their native ISA.

5 Some concrete examples

IBM pioneered the work in the area of virtual machines. In 1964 IBM's new System/360 provided emulation for IBM's older 1401 computers. The VM operating system is another well-known IBM example that introduced virtual machines which used minidisks (virtual disks) and independent OS' in each VM (e.g. CMS). Later IBM emulated 7074 I/O control systems on mainframes like 370 and even IBM PC-DOS on RS/6000. More recent examples include the use of DOS and Windows in the OS/2 environment and the Daisy project (1996).

Microsoft also pioneered some work by creating a CP/M emulator in the late 70-ties.

Another name worth mentioning is Commodore. Although its '86 Amiga computer is considered by many as a toy machine, it was build from the start with emulation in mind. The introduction of the machine was delayed for waiting on a PC-XT/AT emulator to be finished and the '87 'Amiga 2000' model included a set of ISA expansion slots for the sole purpose of being used with their so called bridgeboards. These were near systems-on-a-card with a layer of software called the Janus library managing communication between the host 680x0 and the card's x86.
Due to the lack of any serious amount of business software, the Amiga spawned an enormous amount of emulators. The most notable include the Macintosh emulators Amax (Readysoft), Emplant (Utilities Unlimited) and Shapeshifter (shareware). Emplant was the first to achieve the running of two supervisor-mode operating systems cooperatively, an amazing accomplishment.

Insignia solutions is another important name. The company developed a PC emulator called Softwindows running on Unix and Macintosh systems. The special scheme being used here is that of porting parts of the Windows OS to the native machine. To give some idea of speed, running on a PowerPC 450Mhz, the emulated machine performs –roughly- like a Pentium 200 MMX class PC. Insignia also provided the 16bit DOS and Windows emulation technology used by Windows NT. Recently the company has concentrated on a highly portable Java virtual machine Jeode for use in embedded systems like Philips Nexperia.

CRAY T3D emulator provided a head start in developing high performance applications for the new MPP system running on the existing CRAY parallel vector supercomputers.

Hewlet-Packard shipped in 1987 one of the earliest commercial binary translation systems for HP 3000 to HP Precision Architecture. Recently HP is working on a dynamic translator called Aries, which works from HP Precision Architecture to the coming IA-64 platform.

Digital Equipment Corporation shipped in 1992 a series of translators to the Alpha platform from VAX/VMS, MIPS/Unix, Sparc/Unix and x86/NT. Especially its x86/NT dynamic translator called FX!32 has received much attention and inspired many newer generations of translators.

Apple Computer abandoned in 1994 its 680x0 based architecture in favor of the new PowerPC. These new PowerMacs were shipped with a 680x0 interpreter, which was later updated to use dynamic translation. This emulator was of strategic importance to Apple since even its own operating system depended on it. This way Apple could incrementally upgrade its OS to the new platform. Years later however the MacOS still contained legacy code, and 680x0 processor were supported up until version 9 came out in late 1999.

Sun released the Java language in 1996 that included an execution model. This model involved compiling to byte code that was to be dynamic translated (or as Sun calls it, compiled just in time) at run time. The virtual machine being used for this is still an ongoing topic of research.

Due to increase in PC desktop performance it became viable in 1996 to emulate old Arcade and console entertainment systems. Being very complicated real time multimedia systems, these required an enormous amount of both coding and performance. For example, a famicom system emulator like SNES9x emulating a 65c816 CPU running at about 3Mhz, needs about a Pentium 550Mhz class PC to run at full speed. This is due to a very complicated system design, involving a sound CPU, DSP (for many effects including digital FIR sound filter), variable length cycles, video subsystem capable of a range of special graphic effects etc.
One of the first of these advanced emulators was a NeoGeo emulator. This soon inspired a lot of others. A notable name is NeoRageX. Other examples include System 16 emulator (Sega System 16), Calus (Capcom CPS-1),RAINE (various TAITO boards, including 32bit F3 system), Modeller (Sega system 32, and Sega model 1) and FinalBurn (Capcom CPS-2).

Finally, a very large emulation project called MAME must be noted. This is a huge open-source project being worked on by a few hundred people all over the world. The system utilizes a modular driver system where individuals can supply drivers for specific systems. The system itself consists of an API for general and reusable emulation functionality and a library of components (CPUs, sound chips etc) that can be arbitrarily used by any driver to form a complete system. On top of this, machine specifics of the MAME system itself are isolated in order to facilitate easy porting. MAME has been ported to a large number of systems, including x86 under DOS, Windows, Linux and Beos, PowerPC Macintosh, PowerPC Amiga and even to embedded systems like a digital camera (!).

6 Conclusion

As we have seen binary translation is not an obscure or new technique. Instead it has been

practiced almost for as long as computers exist and is a very large field. So large, that there is much more to say about the subject. We haven't discussed API emulations like WINE, nor binary translation optimized architectures. There's also a legal factor involved and we haven't provided any details about optimization. We could go on.
We do have seen that binary translation is a viable technique for running legacy applications, for running multiple machines in a single environment and for machine abstraction. An import issue remains performance. With some systems demanding in excess of 100 times more computing power than the legacy machine, we can understand there's a lot of room left for improvement.

We also feel that some techniques can be far better understood. Both HLE and some aspects of pure static translation need a lot more study.

7 References

- Digital FX!32: combining emulation and binary translation, Raymond J. Hookway and Mark A. Herdeg.
- UQBT – A resourceable and retargetable binary translator, Cristina Cifuentes, Mike van Emmerik, Computer, IEEE, March 2000, page 60
- Welcome to the opportunities of binary translation, Erik R. Altman, David Kaeli, Yaron Sheffer. Computer, IEEE, March 2000, page 40
- A methodology for performance evaluation of systems with large emulation code, Humayun Khalid (khalid@ibmoto.com)
- An Alpha in PC clothing, Tom Thompson, feb 1996, Byte page 195
- PA-RISC to IA-64 : Transparent execution, no recompilation, Cindy Zheng, Carol Thompson, Computer, IEEE, march 2000, page 47
- Sony's emotionally charged chip, Keith Diefendorff, microprocessor report, April 1999 page 1
- Operating System concepts, fifth edition, Silberschatz Galvin
- Softwindows 98, Marc Hoffman, www.kearney.net/~mhoffman/softwindows98_review.html
- A lot or articles, developer diaries, documentaries etc reachable through: www.vg-networks.com