Introduction to Virtual Machines

J. E. Smith and Ravi Nair

Chapter 1 of *Virtual Machines: Versatile Platforms for Systems and Processes* to be published by Morgan Kaufmann 2005.

Modern computers are among the most advanced human-engineered structures; and they are possible only because of our ability to manage extreme complexity. Computer systems contain many silicon chips, each with hundreds of millions of transistors. These are interconnected and combined with high-speed I/O devices and networking infrastructure to form the platforms upon which software can operate. Operating systems, application programs and libraries, and graphics and networking software all cooperate to provide a powerful environment for data management, education, communication, entertainment, and many other applications.

The key to managing complexity in computer systems is their division into *levels of abstraction* separated by *well-defined interfaces*. Levels of abstraction allow implementation details at lower levels of a design to be ignored or simplified, thereby simplifying the design of components at higher levels. The details of a hard disk, for example that it is comprised of sectors and tracks, are abstracted by the operating system so the disk appears to application software as a set of variable-sized files (see Figure 1). An application programmer can then create, write, and read files, without knowledge of the way the hard disk is constructed and organized.

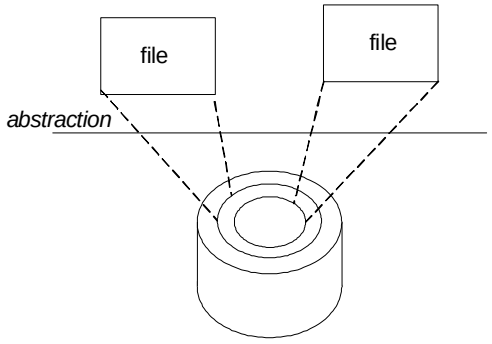

Figure 1. Files are an abstraction of a disk. A level of abstraction provides a simplified interface to underlying resources.

The levels of abstraction are arranged in a hierarchy, with lower levels implemented in hardware and higher levels in software. In the hardware levels, all the components are physical, have real properties, and their interfaces are defined so that the various parts can be physically connected. In the software levels, components are logical, with fewer restrictions based on physical characteristics. In this book, we are most concerned with the abstraction levels that are at or near the hardware/software boundary. These are the levels where software is separated from the *machine* on which it runs.

A *machine* executes computer software, and is a term that dates back to the beginning of computers ("platform" is a term more in vogue today). From the perspective of the operating system, a machine is largely composed of hardware, including one or more processors that run a specific instruction set, some real memory, and I/O devices. However, we do not restrict the use of the term "machine" to just the hardware components of a computer. From the perspective of application programs, for example, the machine is a combination of the operating system and those portions of the hardware accessible through user–level binary instructions.

Let us now turn to the other aspect of managing complexity: the use of *well-defined interfaces*. Well-defined interfaces allow computer design tasks to be decoupled so that teams of hardware and software designers can work more or less independently. The instruction set is one such interface. Processor designers, say at Intel, develop microprocessors that implement the Intel IA-32[1] instruction set, while software engineers at Microsoft develop compilers that map high level languages to the same instruction set. As long as both groups satisfy the instruction set specification, compiled software will execute correctly on a machine incorporating the IA-32 microprocessor. The operating system interface, defined as a set of function calls, is another important standardized interface in computer systems. As the Intel/Microsoft example suggests, well-defined interfaces permit development of interacting computer subsystems at different companies, and at different times, sometimes years apart. Application software developers do not need to be aware of detailed changes inside the operating system, and hardware and software can be upgraded according to different schedules. Software can run on different platforms implementing the same instruction set.

Despite their many advantages, well-defined interfaces can also be confining. Subsystems and components designed to specifications for one interface will not work with those designed for another. There are processors with different instruction sets (e.g., Intel IA-32 and IBM PowerPC), and there are different operating systems (e.g., Windows and Linux). Application programs, when distributed as program binaries, are tied to a specific instruction set and operating system. An operating system is tied to a computer that implements specific memory system and I/O system interfaces. As a general principle, diversity in instruction sets, operating systems, and application programming languages encourages innovation and discourages stagnation. However, in practice, diversity also leads to reduced interoperability, which becomes restrictive, especially in a world of networked computers where it is advantageous to move software as freely as data.

Beneath the hardware/software interface, hardware resource considerations can also limit the flexibility of software systems. Memory and I/O abstractions, both in high level languages and in operating system, have removed many hardware resource dependences; some still remain, however. Many operating systems are developed for a specific system architecture, e.g., for a uniprocessor or a shared memory multiprocessor, and are designed to manage hardware resources directly. The implicit assumption is that the hardware resources of a system are managed by a single operating system. This binds all hardware resources into a single entity under a single management regime. And this, in turn, limits the flexibility of the system, not only in terms of available software (as discussed above), but also in terms of security and failure isolation, especially when the system is shared by multiple users or groups of users.

Virtualization provides a way of relaxing the above constraints and increasing flexibility. When a system (or subsystem), e.g.,., a processor, memory, or I/O device, is *virtualized*, its interface and all resources visible through the interface are mapped onto the interface and resources of a real system actually implementing it. Consequently, the real system is transformed so that it appears to be a different, virtual system, or even a set of multiple virtual systems. Formally, virtualization involves the construction of an isomorphism that maps a virtual *guest* system to a real *host* (Popek and Goldberg 1974). This isomorphism, illustrated in Figure 2, maps the guest state to the host state (function V in Figure 2), and for a sequence of operations that modify the state in the guest (the function e_i modifies state S_i to state S_j) there is a corresponding sequence of operations e' in the host that performs an equivalent modification to the host's state (changes S_i' to S_j'). Although such an isomorphism can be used to characterize abstraction as well as virtualization, we distinguish the two: virtualization differs from abstraction in that virtualization does not necessarily hide details; the level of detail in a virtual system is often the same as that in the underlying real system.

[1] Sometimes referred to informally as "x86".

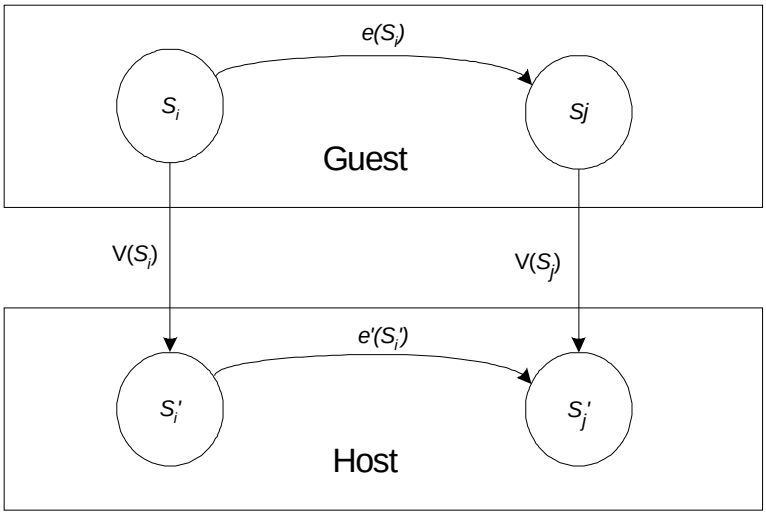

Figure 2. Formally, virtualization is the construction of an isomorphism between a guest system and a host; $e' \circ V(S_i) = V \circ e(S_i)$.

Consider again the example of a hard disk. In some applications, it may be desirable to partition a single large hard disk into a number of smaller virtual disks. The virtual disks are mapped to a real disk by implementing each of the virtual disks as a single large file on the real disk (Figure 3). Virtualizing software provides a mapping between virtual disk contents and real disk contents (the function V in the isomorphism) using the file abstraction as an intermediate step. Each of the virtual disks is given the appearance of having a number of logical tracks and sectors (although fewer than in the large disk). A write to a virtual disk (the function *e* in the isomorphism) is mirrored by a file write and a corresponding real disk write, in the host system (the function *e´* in the isomorphism). In this example, the level of detail provided at the virtual disk interface, i.e., sector/track addressing, remains the same as for a real disk; no abstraction takes place.

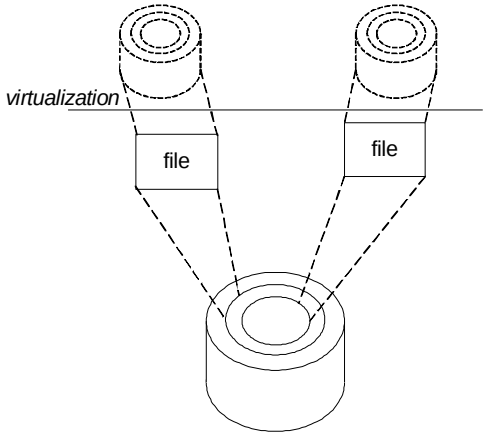

Figure 3. Implementing virtual disks. Virtualization provides a different interface and/or resources at the same level of abstraction.

The concept of virtualization can be applied not only to subsystems such as disks, but to an entire machine. A *Virtual Machine* (VM) is implemented by adding a layer of software to a real machine to support the desired virtual machine's architecture. For example, virtualizing software installed on an Apple MacIntosh can provide a Windows/IA-32 virtual machine capable of running PC application programs. In general, a virtual

machine can circumvent real machine compatibility constraints and hardware resource constraints to enable a higher degree of software portability and flexibility.

There is a wide variety of virtual machines that provide an equally wide variety of benefits. Multiple, replicated virtual machines can be implemented on a single hardware platform to provide individuals or user groups with their own operating system environments. The different system environments (possibly with different operating systems), also provide isolation and enhanced security. A large multiprocessor server can be divided into smaller, virtual servers while retaining the ability to balance the use of hardware resources across the system.

Virtual machines can also employ emulation techniques to support cross-platform software compatibility. For example, a platform implementing the PowerPC instruction set can be converted into a virtual platform running the IA-32 instruction set. Consequently, software written for one platform will run on the other. This compatibility can be provided either at the system level (e.g., to run Windows OS on a MacIntosh) or at the program or process level (e.g., to run Excel on a Sun Solaris/SPARC platform). In addition to emulation, virtual machines can also provide dynamic, on-the-fly optimization of program binaries. Finally, through emulation, virtual machines can enable new, proprietary instruction sets, e.g., incorporating VLIWs, while supporting programs in an existing, standard instruction set.

The virtual machine examples just described are constructed to match the architectures of existing real machines. However, there are also virtual machines for which there are no corresponding real machines. It has become common for language developers to invent a virtual machine tailored to a new high level language. Programs written in the high level language are compiled to "binaries" targeted at the virtual machine. Then, any real machine on which the virtual machine is implemented can run the compiled code. The power of this approach has been clearly demonstrated with the Java high level language and the Java virtual machine, where a high degree of platform independence has been achieved, thereby enabling a very flexible network computing environment.

Virtual machines have been investigated and built by operating system developers, language designers, compiler developers, and hardware designers. Although each application of virtual machines has its unique characteristics, there also are underlying concepts and technologies that are common across the spectrum of virtual machines. Because the various virtual machine architectures and underlying technologies have been developed by different groups it is especially important to unify this body of knowledge and understand the base technologies that cut across the various forms of virtual machines. The goals of this book are to describe the family of virtual machines in a unified way, to discuss the common underlying technologies that support them, and to demonstrate their versatility by exploring their many applications.

1.1 Computer Architecture

As will become evident, a discussion of virtual machines is also a discussion about computer architecture in a broad sense. Virtual machines often bridge architectural boundaries, and a major consideration in constructing a virtual machine is the fidelity with which a virtual machine implements architected interfaces. Therefore, it is useful to define and summarize "computer architecture".

The architecture of a building describes its functionality and appearance to users of the building, but not the details of its implementation, such as the specifics of its plumbing system or the manufacturer of bricks used in its construction. Analogously, the term *architecture*, when applied to computers, refers to the functionality and appearance of a computer system or subsystem, but not the details of its construction. The architecture is often formally described through a specification of an interface and the logical behavior of resources ma-

nipulated via the interface. The term *implementation* will be used to describe the actual embodiment of an architecture. Any architecture can have several implementations, each one having distinct characteristics, e.g., a high-performance implementation or a low-power implementation.

The levels of abstraction in a computer system correspond to *implementation layers* in both hardware and software. There is an architecture for each of these implementation layers (although the term "architecture" is not always used). Figure 4 shows some of the important interfaces and implementation layers as found in a typical computer system. In software, for example, there is an interface between an application program and standard libraries (interface 2 in Figure 4). Another software interface is at the boundary of the operating system (interface 3). The interfaces in hardware include an I/O architecture that describes the signals that drive I/O device controllers (interface 11), a hardware memory architecture that describes the way addresses are translated (interface 9), an interface for the memory access signals that leave the processor (interface 12), and another for the signals that reach the DRAM chips in memory (interface 14). The OS communicates with I/O devices through a sequence of interfaces: 4, 8, 10, 11, and 13. Of these interfaces and architectures, we are most interested in those at or near the hardware/software boundary.

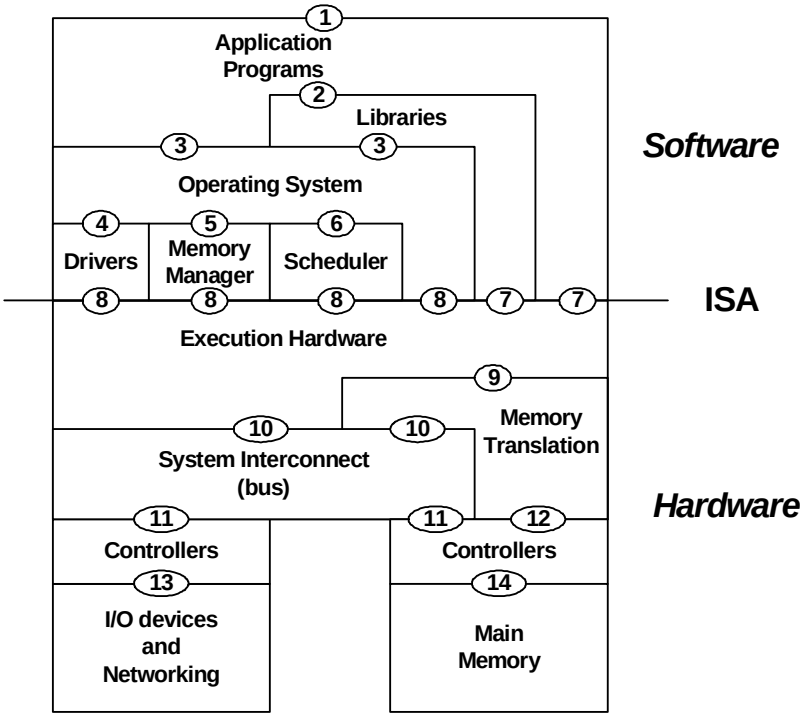

Figure 4. Computer system architectures; Implementation layers communicate vertically via the shown interfaces. This view of architecture is styled after one given by Glenford Myers (Myers 1982).

The *Instruction Set Architecture* (ISA) marks the division between hardware and software, and is composed of interfaces 7 and 8 in Figure 4. The concept of an ISA was first clearly articulated when the IBM 360 family of mainframe computers was developed in the early 1960s (Amdahl, Blaauw, and Brooks 1964). With that project, the importance of software compatibility was fully recognized. The IBM 360 series had a number of models that could incorporate a wide range of hardware resources, thereby covering a broad spectrum of price and performance levels -- yet they were designed to run the same software. To successfully accomplish

this, the interface between the hardware and software had to be precisely defined and controlled, and the ISA serves this purpose.

There are two parts of an ISA that are important in the definition of virtual machines. The first part includes those aspects of the ISA that are visible to an application program. This will be referred to as the *user ISA*. The second part includes those aspects that are visible only to supervisor software, such as the operating system, which is responsible for managing hardware resources. This is the *system ISA*. Of course, the supervisor software can also employ all the elements of the user ISA. In Figure 4, interface 7 consists of the user ISA only, and interface 8 consists of both the user and system ISA.

In this book we will also be concerned with interfaces besides the ISA.. The Application Binary Interface (ABI), consists of interfaces 3 and 7 in Figure 4. An important related interface is the Application Program Interface (API), which consists of interfaces 2 and 7.

The *Application Binary Interface* (ABI) provides a program with access to the hardware resources and services available in a system and has two major components. The first is the set of all user instructions (interface 7 in Figure 4); system instructions are not included in the ABI. At the ABI level, all application programs interact with the shared hardware resources indirectly, by invoking the operating system via a *system call* interface (interface 3 in Figure 4), which is the second component of the ABI. System calls provide a specific set of operations that an operating system may perform on behalf of a user program (after checking to make sure that the user program should be granted its request). The system call interface is typically implemented via an instruction that transfers control to the operating system in a manner similar to a procedure or subroutine call, except the call target address is forced to be a specific address in the operating system. Arguments for the system call are typically passed through registers and/or a stack held in memory, following specific conventions that are part of the system call interface. A program binary compiled to a specific ABI can run unchanged only on a system with the same ISA and operating system.

The *Application Programming Interface* (API) is usually defined with respect to a high level language (HLL). A key element of an API is a standard library (or libraries) that an application calls to invoke various services available on the system, including those provided by the operating system. An API is typically defined at the source code level and enables applications written to the API to be ported easily (via recompilation) to any system that supports the same API. The API specifies an abstraction of the details of implementation of services, especially those that involve privileged hardware. For example, `clib` is a well-known library that supports the UNIX/C programming language. The `clib` API provides a memory model consisting of text (for code), and a heap and stack (for data). A routine belonging to an API typically contains one or more ABI-level operating system calls. Some API library routines are simply "wrappers", i.e., code that translates directly from the HLL calling convention to the binary convention expected by the OS. Other API routines are more complex, and may include several OS calls.

1.2 Virtual Machine Basics

To understand what a "Virtual Machine" is, we must first discuss what is meant by "machine", and, as pointed out above, the meaning of "machine" is a matter of perspective. From the perspective of a *process* executing a user program, the machine consists of a logical memory address space that has been assigned to the process, along with user level registers and instructions that allow the execution of code belonging to the process. The I/O part of the machine is visible only through the operating system, and the only way the process can interact with the I/O system is via operating system calls, often through libraries that execute as part of the process. Processes are often transient in nature (although not always). They are created, execute for a period of time, perhaps spawn other processes along the way, and eventually terminate. To summarize,

the machine, from the prospective of a process, is a combination of the operating system and the underlying user-level hardware. The ABI provides the interface between the process and the machine (Figure 5a).

From the perspective of the operating system, an entire *system* is supported by the underlying machine. A system is a full execution environment that can simultaneously support a number of processes potentially belonging to different users. All the processes share a file system and other I/O resources. The system environment persists over time (with occasional re-boots) as processes come and go. The system allocates real memory and I/O resources to the processes, and allows the processes to interact with their resources via an OS that is part of the system. Hence, the machine, from the perspective of a system, is implemented by the underlying hardware alone, and the ISA provides the interface between the system and the machine (Figure 5b).

In practical terms, a virtual machine executes software (either an individual process or a full system, depending on the type of machine) in the same manner as the machine for which the software was developed. The virtual machine is implemented as a combination of a real machine and virtualizing software. The virtual machine may have resources different from the real machine, either in quantity or in type. For example, a virtual machine may have more or fewer processors than the real machine, and the processors may execute a different instruction set than the real machine. It is important to note that equivalent performance is usually not required as part of virtualization; often a virtual machine provides less performance than an equivalent real machine running the same software, i.e., software developed for the real machine.

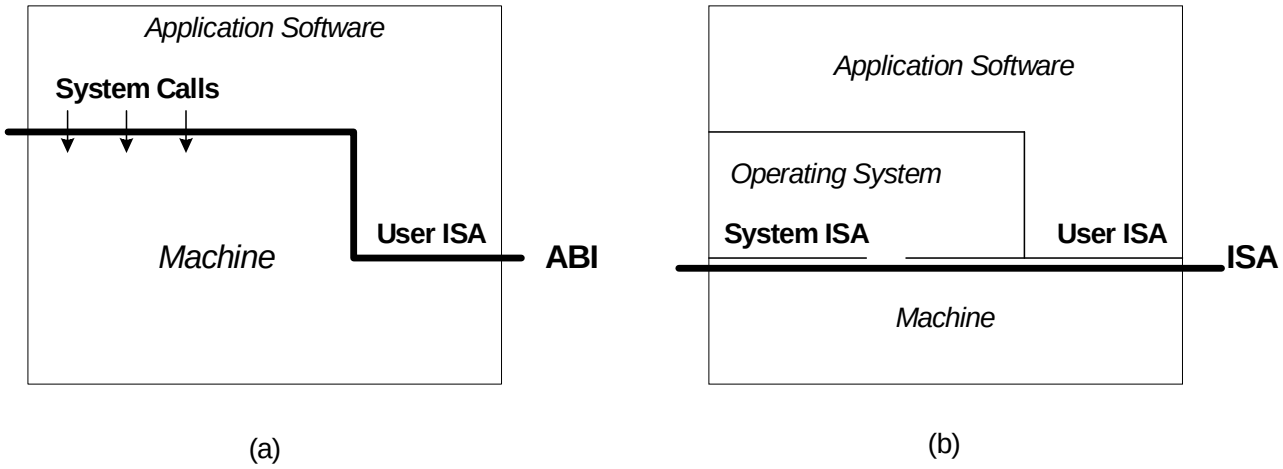

Figure 5. Machine Interfaces a)
Application Binary Interface (ABI)
b) Instruction Set Architecture (ISA) interface

As characterized by the isomorphism described earlier, the process of virtualization consists of two parts 1) the mapping of virtual resources or state, e.g., registers, memory, or files, to real resources in the underlying machine and 2) the use of real machine instructions and/or system calls to carry out the actions specified by virtual machine instructions and/or system calls; e.g., emulation of the virtual machine ABI or ISA.

Just as there is a process perspective and a system perspective of machines, there are also process level and system level virtual machines. As the name suggests, a *process virtual machine* is capable of supporting an individual process. A process virtual machine is illustrated in Figure 6. In this figure and in figures that follow, compatible interfaces are illustrated graphically as "meshing" boundaries. In process VMs, the virtualiz-

ing software is placed at the ABI interface, on top of the OS/hardware combination .The virtualizing software emulates both user level instructions and operating system calls.

With regard to terminology (see Figure 6), we usually refer to the underlying platform as the *host,* and the software that runs in the VM environment as the *guest*. The real platform that corresponds to a virtual machine, i.e., the real machine being emulated by the virtual machine, is referred to as the *native* machine. The name given to the virtualizing software depends on the type of virtual machine being implemented. In process VMs, virtualizing software is often referred to as the *runtime,* which is short for "runtime software"[2]. The runtime is created to support a guest process and runs on top of an operating system. The VM supports the guest process as long as the guest process executes, and terminates support when the guest process terminates.

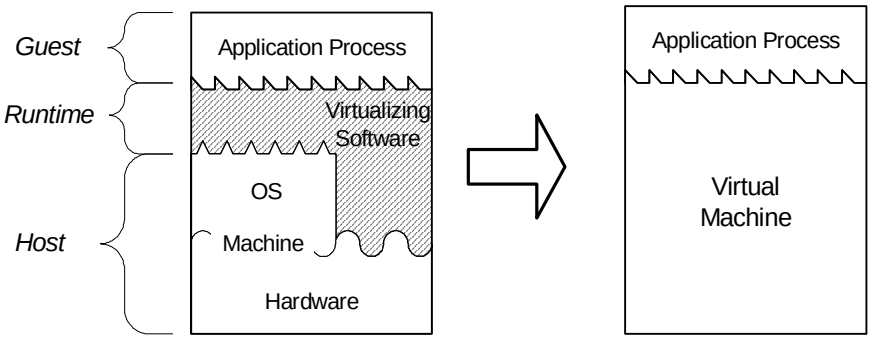

Figure 6. Virtualizing software translates a set of OS and user level instructions composing one platform to another, forming a Process Virtual Machine capable of executing programs developed for a different OS and different ISA.

In contrast, a *system virtual machine* provides a complete system environment. This environment can support an operating system along with its potentially many user processes. It provides a guest operating system with access to underlying hardware resources, including networking, I/O, and, on the desktop, a display and graphical user interface. The VM supports the operating system as long as the system environment is alive.

A system virtual machine is illustrated in Figure 7; virtualizing software is placed between the underlying hardware machine and conventional software. In this particular example, virtualizing software emulates the hardware ISA so that conventional software "sees" a different ISA than the one supported by hardware. In many system VMs the guest and host run the same ISA, however. In system VMs, the virtualizing software is often referred to as the *Virtual Machine Monitor* (VMM), a term coined when the VM concept was first developed in the late 1960's.

[2] Throughout this book, we will use the compound form "runtime" as a noun to describe the virtualizing runtime software in a process VM; "run time" will be used in the more generic sense: the time during which a program is running.

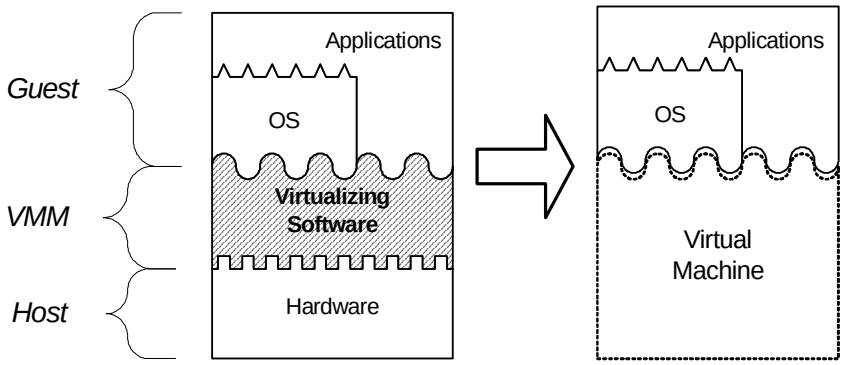

Figure 7 Virtualizing software translates the ISA used by one hardware platform to another, forming a System Virtual Machine, capable of executing a system software environment developed for a different set of hardware.

Virtualizing software can be applied in several ways to connect and adapt the major computer subsystems (see Figure 8). *Emulation* adds considerable flexibility by permitting "mix and match" cross-platform software portability. In this example (Figure 8a), one ISA is emulated by another. Virtualizing software can enhance emulation with *optimization,* by taking implementation-specific information into consideration as it performs emulation. Virtualizing software can also provide resource *replication,* for example by giving a single hardware platform the appearance of multiple platforms (Figure 8b), each capable of running a complete operating system and/or a set of applications. Finally, the virtual machine functions can be composed (Figure 8c) to form a wide variety of architectures, freed of many of the traditional compatibility and resource constraints.

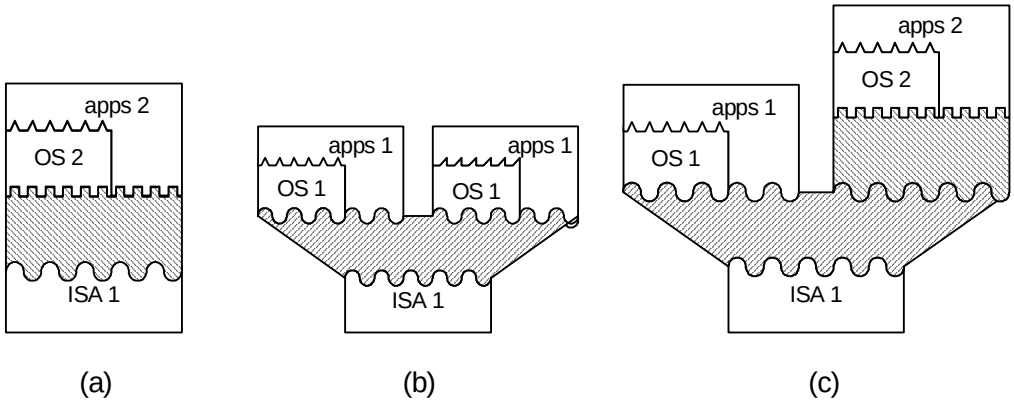

Figure 8. Examples of virtual machine applications.

a) emulating one instruction set with another
b) replicating a virtual machine so that multiple OSes can be supported simultaneously,
c) composing virtual machine software to form a more complex, flexible system.

We are now ready to describe some specific types of virtual machines. These span a fairly broad spectrum of applications, and we will discuss them according to the two main categories: Process VMs and System VMs. Note that because the various virtual machines have been developed by different design communities, they often use different terms for describing similar concepts and features. In fact, it is sometimes the practice to use some term other than "virtual machine" to describe what is in reality a form of virtual machine.

1.3 Process Virtual Machines

Process level VMs provide user applications with a virtual ABI environment. In their various implementations, process VMs can provide replication, emulation, and optimization. The following subsections describe each of these.

1.3.1 Multiprogramming

The first and most common virtual machine is so ubiquitous that we don't even think of it as being a virtual machine. The combination of the OS call interface and the user instruction set form the machine that executes a user process. Most OSes can simultaneously support multiple user processes through multiprogramming where each user process is given the illusion of having a complete machine to itself. Each process is given its own address space and is given access to a file structure. The operating system timeshares the hardware and manages underlying resources to make this possible. In effect, the operating system provides a replicated process level virtual machine for each of the concurrently executing applications.

1.3.2 Emulators and Dynamic Binary Translators

A more challenging problem for process level virtual machines is to support program binaries compiled to a different instruction set than the one executed by the host's hardware, i.e., to *emulate* one instruction set on hardware designed for another. An example emulating process virtual machine is illustrated in Figure 9. Application programs are compiled for a *source ISA*, but the hardware implements a different *target ISA*. As shown in the example, the operating system may be the same for both the guest process and the host platform, although in other cases the OSes may differ as well. The example in Figure 9 illustrates the Digital FX!32 system (Hookway and Herdeg 1997). The FX!32 system can run Intel IA-32 application binaries compiled for Windows NT on an Alpha hardware platform also running Windows NT.

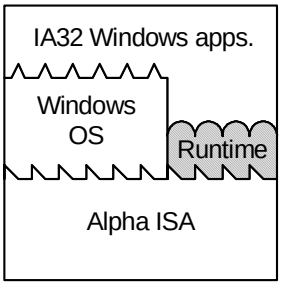

Figure 9. A process VM that emulates guest applications. The Digital FX!32 system allows Windows IA-32 applications to be run on an Alpha Windows platform.

The most straightforward emulation method is *interpretation*. An interpreter program executing the target ISA fetches, decodes and emulates the execution of individual source instructions. This can be a relatively slow process, requiring tens of native target instructions for each source instruction interpreted.

For better performance, *binary translation* is typically used. With binary translation, blocks of source instructions are converted to target instructions that perform equivalent functions. There can be a relatively high overhead associated with the translation process, but once a block of instructions is translated, the translated instructions can be cached and repeatedly executed much faster than they can be interpreted. Because binary translation is the most important feature of this type of process virtual machine, they are sometimes called *dynamic binary translators*.

Interpretation and binary translation have different performance characteristics. Interpretation has relatively low startup overhead but consumes significant time whenever an instruction is emulated. On the other hand, binary translation has high initial overhead when performing the translations, but it is fast for each repeated execution. Consequently, some virtual machines use a staged emulation strategy combined with *profiling*, i.e., the collection of statistics regarding the program's behavior. Initially, a block of source instructions is interpreted, and profiling is used to determine which instruction sequences are frequently executed. Then, a frequently executed block may be binary translated. Some systems perform additional code optimizations on the translated code if profiling shows that it has a very high execution frequency. In most emulating virtual machines the stages of interpretation and binary translation can both occur over the course of a single program's execution. In the case of FX!32, translation occurs incrementally between program runs.

1.3.3 Same-ISA Binary Optimizers

Most dynamic binary translators not only translate from source to target code, but they also perform some code optimizations. This leads naturally to virtual machines where the instruction sets used by host and the guest are the same, and optimization of a program binary is the primary purpose of the virtual machine. Thus, *same-ISA dynamic binary optimizers* are implemented in a manner very similar to emulating virtual machines, including staged optimization and software caching of optimized code. Same-ISA dynamic binary optimizers are most effective for source binaries that are relatively unoptimized to begin with, a situation that is fairly common in practice. A dynamic binary optimizer can collect a profile, and then use this profile information to optimize the binary code on-the-fly. An example of such a same-ISA dynamic binary optimizer is the Dynamo system, originally developed as a research project at Hewlett-Packard (Bala, Duesterwald, and Banerjia 2000).

1.3.4 High Level Language VMs: Platform Independence

For the process VMs described above, cross-platform portability is clearly a very important objective. For example, the FX!32 system enabled portability of application software compiled for a popular platform (IA-32 PC) to a less popular platform (Alpha). However, this approach allows cross-platform compatibility on a case-by-case basis and requires a great deal of programming effort. For example, if one wanted to run IA-32 binaries on all the hardware platforms currently in use, e.g., SPARC, PowerPC, MIPS, etc., then an FX!32-like VM would have to be developed for each of them. The problem would be even more difficult if the host platforms run different OSes than the one for which binaries were originally compiled.

Full cross-platform portability is more easily achieved by taking a step back and designing it into an overall software framework. One way of accomplishing this is to design a process level VM at the same time as an application development environment is being defined. Here, the VM environment does not directly correspond to any real platform. Rather it is designed for ease of portability and to match the features of a high level language (HLL) used for application program development. These *High Level Language VMs* (HLL VMs) are similar to the process VMs described above. However, they are focused on minimizing hardware-specific and OS-specific features because these would compromise platform independence.

HLL VMs first became popular with the Pascal programming environment (Bowles 1980). In a conventional system, Figure 10a, the compiler consists of a front-end that performs lexical, syntax, and semantic analysis to generate simple intermediate code – similar to machine code, but more abstract. Typically the intermediate code does not contain specific register assignments, for example. Then, a code generator takes the intermediate code and generates a binary containing machine code for a specific ISA and OS. This binary file is distributed and executed on platforms that support the given ISA/OS combination. To execute the program on a different platform, however, it must be re-compiled for that platform.

In HLL VMs, this model is changed (Figure 10b). The steps are similar to the conventional ones, but the point at which program distribution takes place is at a higher level. As shown in Figure 10b, a conventional compiler front-end generates abstract machine code, which is very similar to an intermediate form. In many HLL VMs, this is a rather generic stack-based ISA. This *virtual ISA* is in essence the machine code for a virtual machine. The portable virtual ISA code is distributed for execution on different platforms. For each platform, a VM capable of executing the virtual ISA is implemented. In its simplest form, the VM contains an interpreter that takes each instruction, decodes it, and then performs the required state transformations (e.g., involving memory and the stack). I/O functions are performed via a set of standard library calls that are defined as part of the VM. In more sophisticated, higher performance VMs, the abstract machine code may be compiled (binary translated) into host machine code for direct execution on the host platform.

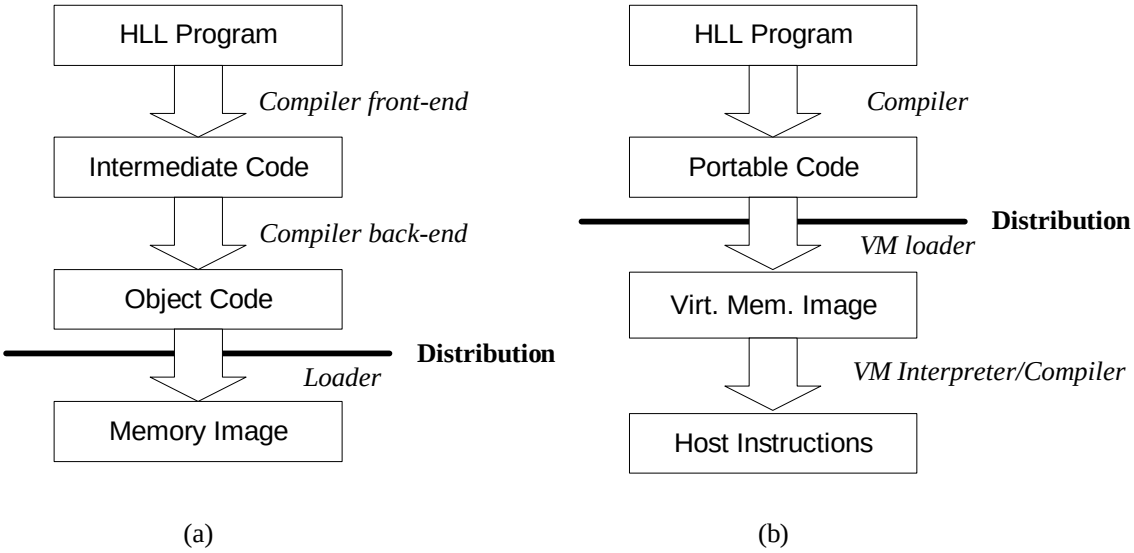

Figure 10. High Level Language environments
a) A conventional system where platform-dependent object code is distributed
b) A HLL VM environment where portable intermediate code is "executed" by a platform-dependent virtual machine.

An advantage of a HLL VM is that software is easily portable, once the VM is implemented on a target platform. While the VM implementation would take some effort, it is a much simpler task than developing a compiler for each platform and re-compiling every application when it is ported. It is also much simpler than developing a conventional emulating process VM for a typical real-world ISA.

The Sun Microsystems Java VM architecture (Lindholm and Yellin 1996) and the Microsoft Common Language Infrastructure (CLI), which is the foundation of the .NET framework (Box 2002), are more recent, widely used examples of HLL VMs. Platform independence and high security are central to both the Java VM and CLI. The ISAs in both systems are based on *bytecodes*, that is, instructions are encoded as a sequence of bytes, where each byte is an opcode, a single-byte operand, or part of a multi-byte operand. These bytecode instruction sets are stack-based (to eliminate register requirements) and have an abstract data specification and memory model. In fact, the memory size is conceptually unbounded, with garbage collection as an assumed part of the implementation. Because all hardware platforms are potential targets for executing Java or CLI-based programs, applications are not compiled for a specific OS. Rather, a set of standard libraries is provided as part of the overall execution environment.

1.4 System Virtual Machines

System virtual machines provide a complete system environment in which many processes, possibly belonging to multiple users, can co-exist. These VMs were first developed during the 1960s and early 1970s, and they were the origin of the term "virtual machine". By using system VMs, a single host hardware platform can support multiple guest OS environments simultaneously. At the time they were first developed, mainframe computer systems were very large and expensive, and computers were almost always shared among a large number of users. Different groups of users often wanted different OSes to be run on the shared hardware, and VMs allowed them to do so. Alternatively, a multiplicity of single-user OSes allowed a convenient way of implementing time-sharing amongst many users. Over time, as hardware became much less expensive, and much of it migrated to the desktop, interest in these classic system VMs faded.

Today, however, system VMs are enjoying renewed popularity. This is partly due to modern-day variations of the traditional motivations for system VMs. The large, expensive mainframe systems of the past are now servers or server farms, and these servers may be shared by a number of users or user groups. Perhaps the most important feature of today's system VMs is that they provide a secure way of partitioning major software systems that run concurrently on the same hardware platform. Software running on one guest system is isolated from software running on other guest systems. Furthermore, if security on one guest system is compromised or if the guest OS suffers a failure, the software running on other guest systems is not affected. The ability to support different operating systems simultaneously, e.g., Windows and Linux (as illustrated in Figure 11), is another reason for their appeal, although it is probably of secondary importance to most users.

In system VMs, platform replication is the major feature provided by a VMM. The central problem is that of dividing a single set of hardware resources among multiple guest operating system environments. The VMM has access to, and manages all the hardware resources. A guest operating system and application programs compiled for that operating system are then managed under (hidden) control of the VMM. This is accomplished by constructing the system so that when a guest OS performs certain operations, such as a privileged instruction that directly involves the shared hardware resources, the operation is intercepted by the VMM, checked for correctness, and performed by the VMM on behalf of the guest. Guest software is unaware of the "behind the scenes" work performed by the VMM.

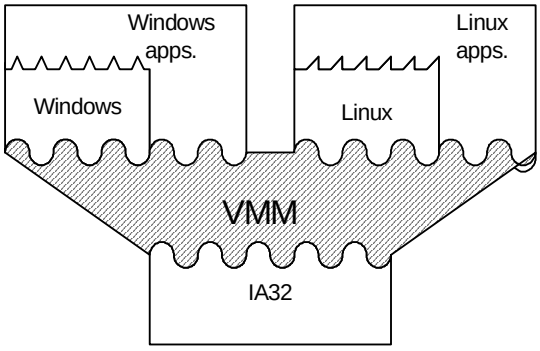

Figure 11. A System VM that supports multiple OS environments on the same hardware.

1.4.1 Implementations of System VMs

From the user perspective, most system VMs provide more-or-less the same functionality. The thing that tends to differentiate them is the way in which they are implemented. As discussed above in Section 1.2,

there are a number of interfaces in a computer system, and this leads to a number of choices for locating the system VMM software. Summaries of two of the more important implementations follow.

Figure 11 illustrates the classic approach to system VM architecture (Popek and Goldberg 1974). The VMM is first placed on bare hardware, and virtual machines fit on top. The VMM runs in the most highly privileged mode, while all the guests systems run with lesser privileges. Then, in a completely transparent way, the VMM can intercept and implement all the guest OS's actions that interact with hardware resources. In many respects, this system VM architecture is the most efficient, and provides service to all the guest systems in a more-or-less equivalent way. One disadvantage of this type of system, at least for desktop users, is that installation requires wiping an existing system clean and starting from scratch, first installing the VMM, and then installing guest OSes on top. Another disadvantage is that I/O device drivers must be available for installation in the VMM, because it is the VMM that interacts directly with I/O devices.

An alternative system VMM implementation builds virtualizing software on top of an existing host operating system – resulting in what is called a *hosted VM*. With a hosted VM, the installation process is similar to installing a typical application program. Furthermore, virtualizing software can rely on the host OS to provide device drivers and other lower-level services; they don't have to be provided by the VMM. The disadvantage of this approach is that there can be some loss of efficiency because more layers of software become involved when OS service is required. The hosted VM approach is taken in the VMware implementation (VMware 2000), a modern VM that runs on IA-32 hardware platforms.

1.4.2 Whole System VMs: Emulation

In the conventional system VMs described above, all the system software (both guest and host) and application software use the same ISA as the underlying hardware. In some important situations, however, the host and guest systems do not have a common ISA. For example, the Apple PowerPC-based systems and Windows PC use different ISAs (and different OSes), and they are the two most popular desktop systems today. As another example, Sun Microsystems servers use a different OS and ISA than the Windows PCs that are commonly attached to them as clients. Because software systems are so closely tied to hardware systems, this may require purchase of multiple platform types, even when unnecessary for any other reason, it complicates software support, and/or it restricts the availability of useful software packages to users.

This situation motivates system VMs where a complete software system, both OS and applications, are supported on a host system that runs a different ISA and OS. These are called *whole system VMs* because they essentially virtualize all software. Because the ISAs are different, both application and OS code require emulation, e.g., via binary translation. For whole system VMs, the most common implementation method is to place the VMM and guest software on top of a conventional host OS running on the hardware.

Figure 12 illustrates a whole system VM built on top of a conventional system with its own OS and application programs. An example of this type of VM is Virtual PC (Traut 1997), which enables a Windows system to run on a MacIntosh platform. The VM software executes as an application program supported by the host OS and uses no system ISA operations. It is as if the VM software, the guest OS and guest application(s) are one very large application implemented on the host OS and hardware. Meanwhile the host OS can also continue to run applications compiled for the native ISA; this feature is illustrated in the right section of the drawing.

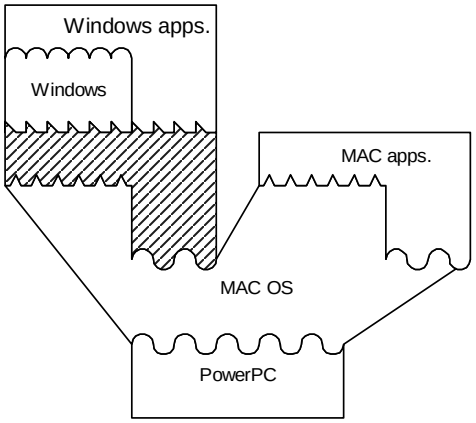

Figure 12. A whole system VM that supports a guest OS and applications, in addition to host applications.

To implement a system VM of this type, the VM software must emulate the entire hardware environment. It must control the emulation of all the instructions, and it must convert the guest system ISA operations to equivalent OS calls made to the host OS. Even if binary translation is used, it is tightly constrained because translated code often cannot take advantage of underlying system ISA features like virtual memory management and trap handling. In addition, problems can arise if the properties of hardware resources are significantly different in the host and the guest. Solving these mismatches is the major challenge of implementing whole system VMs.

1.4.3 Co-Designed VMs: Hardware Optimization

In all the VM models discussed thus far, the goal has been functionality and portability -- to either support multiple (possibly different) OSes on the same host platform, or to support different ISAs and OSes on the same platform. In practice, these virtual machines are implemented on hardware already developed for some standard ISA, and for which native (host) applications, libraries, and operating systems already exist. By-and-large, improved performance (i.e., going beyond native platform performance) has not been a goal -- in fact minimizing performance losses is often the performance goal.

Co-Designed VMs have a different objective and take a different approach. These VMs are designed to enable innovative ISAs and/or hardware implementations for improved performance, power efficiency, or both. The host's ISA may be completely new, or it may be based on an existing ISA with some new instructions added and/or some instructions deleted. In a co-designed VM, there are no native ISA applications. It is as if the VM software is, in fact, part of the hardware implementation.

In some respects, co-designed virtual machines are similar to a purely hardware virtualization approach used in many high performance superscalar microprocessors. In these designs, hardware renames architected registers to a larger number of physical registers, and complex instructions are decomposed into simpler, RISC-like instructions (Hennessy and Patterson 2002). In this book, however, we focus on software implemented co-designed virtual machines; binary translation is over a larger scope and can be more flexible because it is done in software.

Because the goal is to provide a VM platform that looks exactly like a native hardware platform, the software portion of a co-designed VM uses a region of memory that is not visible to any application and system software. This concealed memory is carved out of physical memory at boot time and the conventional guest software is never informed of its existence. VMM code that resides in the concealed memory can take con-

trol of the hardware at practically any time and perform a number of different functions. In its more general form, the VM software includes a binary translator that converts guest instructions into native ISA instructions and caches the translated instructions in a region of concealed memory. Hence, the guest ISA never directly executes on the hardware. Of course, interpretation can also be used to supplement binary translation, depending on performance tradeoffs. To provide improved performance, translation is coupled with code optimization. Optimization of frequently executed code sequences is performed at translation time, and/or it is performed as an on-going process while the program runs.

Perhaps the best-known example of a co-designed VM is the Transmeta Crusoe (Halfhill 2000). In this processor, the underlying hardware uses a native VLIW instruction set, and the guest ISA is the Intel IA-32. In their implementation, the Transmeta designers focused on the power saving advantages of simpler VLIW hardware. An important computer system that relies on many co-designed VM techniques is the IBM AS/400 system (Soltis 1996). The AS/400 differs from the other co-designed VMs, because the primary design objective is support for an object-oriented instruction set that re-defines the hardware/software interface in a novel fashion. The current AS/400 implementations are based on an extended PowerPC ISA, although earlier versions used a considerably different, proprietary CISC ISA.

1.5 A Taxonomy and Summary

We have just described a rather broad array of VMs, with different goals and implementations. To put them in perspective and organize the common implementation issues, we introduce a taxonomy illustrated in Figure 13. First, VMs are divided into the two major types: Process VMs and System VMs. In the first type, the VM supports an ABI – user instructions plus system calls, in the second, the VM supports a complete ISA – both user and system instructions. Finer divisions in the taxonomy are based on whether the guest and host use the same ISA.

On the left side of Figure 13 are process VMs. These include VMs where the host and guest instruction sets are the same. In the figure, we identify two examples. The first is multiprogrammed systems, as already supported on most of today's systems. The second is same-ISA dynamic binary optimizers, which transform guest instructions only by optimizing them, and then execute them natively.

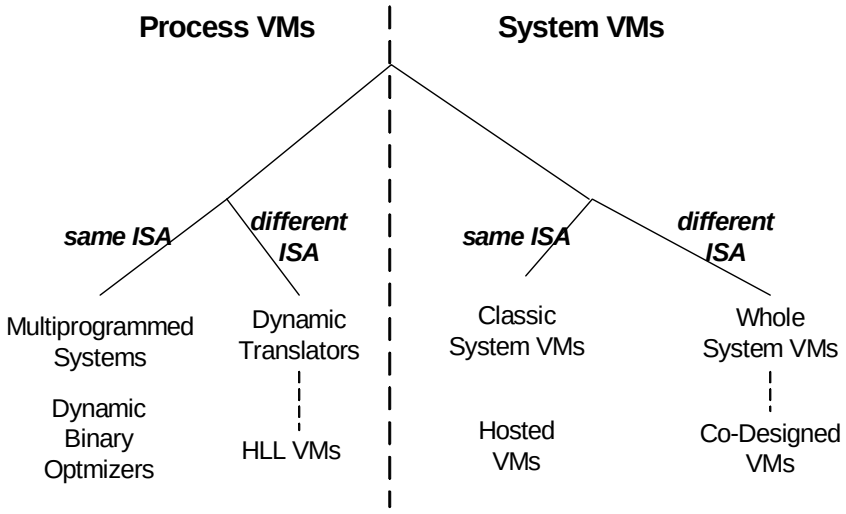

Figure 13. A taxonomy of virtual machines.

For process VMs where the guest and host ISAs are different, we also give two examples. These are dynamic translators and HLL VMs. HLL VMs are connected to the VM taxonomy via a "dotted line" because their process level interface is at a different, higher level than the other process VMs.

On the right side of the figure are system VMs. If the guest and host use the same ISA, examples include "classic" System VMs and Hosted VMs. In these VMs, the objective is providing replicated, isolated system environments. The primary difference between classic and hosted VMs is the VMM implementation, rather the function that is provided to the user.

Examples of system VMs where the guest and host ISAs are different include Whole System VMs and Co-Designed VMs. With Whole System VMs, performance is often of secondary importance compared to accurate functionality, while with Co-Designed VMs, performance (and power efficiency) are often major goals. In the figure, Co-Designed VMs are connected using dotted lines because their interface is typically at a lower level than other system VMs.

THE VERSATILITY OF VIRTUAL MACHINES

A good way to summarize this chapter is with an example of a realistic system that could conceivably be in use today (Figure 14). The example clearly illustrates the versatility of virtual machine technologies. A computer user might have a Java application running on a laptop PC. This is nothing special; it is done via a Java virtual machine developed for IA-32/Linux. However, the user happens to have Linux installed as an OS VM via VMware executing on a Windows PC. And, as it happens, the IA-32 hardware is in fact a Transmeta Crusoe, a co-designed VM implementing a VLIW ISA with binary translation (what Transmeta calls "code morphing") to support the IA-32 ISA. By using the many VM technologies, a Java bytecode program is actually executing as native VLIW.

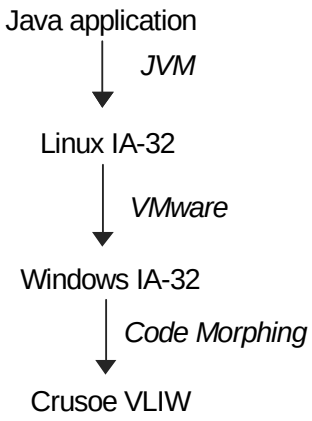

Figure 14. Three levels of VMs: a Java application running on a Java VM, running on an system VM, running on a co-designed VM.

THE REST OF THE BOOK

The book can be divided into two major sections, along the lines of the VM taxonomy described above. Chapters 2 through 6 deal primarily with process virtual machines, and chapters 7 through 9 deal primarily with system virtual machines.

It should now be evident that instruction set emulation is an important enabling technology for many virtual machine implementations. Because of its importance, we begin, in Chapter 2, with a detailed discussion of

emulation. Emulation encompasses both interpretation, where guest instructions are emulated one-by-one in a simple fashion, and binary translation, where blocks of guest instructions are translated to the ISA of the host platform and are saved for multiple executions. In many of the VM implementations, the need to use emulation is obvious (i.e., the guest and host ISAs are different), but as we will see, the same techniques are important in other VMs for non-obvious reasons (even in cases where the guest and host ISAs are the same). A case study is the Shade simulation system, which incorporates a number of emulation techniques.

Chapter 3 uses emulation as a starting point and describes the overall architecture and implementation of process VMs. Included are the management of cached binary translations and the handling of complicating issues such as precise traps and self-modifying code. The DEC/Compaq FX!32 system is used as a case study. The FX!32 system supports Windows/IA-32 guest binaries on a Windows/Alpha platform.

Performance is almost always an issue in VM implementations because performance is often lost during the emulation process. This loss can be mitigated by optimizing translated binaries, and Chapter 4 deals with ways of dynamically optimizing translated code. First, ways of increasing the size of translated code blocks are discussed, and then specific code optimizations are covered. Code optimizations include re-ordering instructions, to improve pipeline efficiency for example, and a number of classical compiler optimizations, adapted to dynamically translated binary code. A special case occurs when the source and target ISA are the same and optimization is the primary function provided by the VM. Consequently, Chapter 4 concludes with a discussion of special-case features of these same-ISA dynamic binary optimizers and includes the HP Dynamo system as a case study.

Chapters 5 and 6 discuss high level language virtual machines. These virtual machines are designed to enable platform-independence. To give some historical perspective, Chapter 5 begins with a description of Pascal P-code. Because modern HLL VMs are intended to support network-based computing and object-oriented programming, the features important for supporting these aspects are emphasized. The Java virtual machine architecture is described in some detail. Also included is a shorter discussion of the Microsoft Common Language Infrastructure (CLI); where the goals and applications are somewhat broader than with Java, and the discussion is focused on those features that provide the added breadth. Chapter 6 describes the implementation of HLL VMs, beginning with basic implementation approaches and techniques. High performance HLL VM implementations are then described, and the IBM Jikes research virtual machine is used as a case study.

Co-designed VMs, the first system level virtual machines to be discussed, are the topic of Chapter 7. The co-designed VM paradigm includes special hardware support to enhance emulation performance. Consequently, much of the chapter focuses on hardware-based performance enhancements. The chapter includes case studies of the Transmeta Crusoe and IBM AS/400 system.

Chapter 8 covers conventional system VMs – that is, VMs that support multiple OSes simultaneously, primarily relying on software techniques. Basic mechanisms for implementing system VMs and for enhancing their performance are discussed. Some ISAs are easier to virtualize than others, so features of instruction sets that make them more efficiently virtualized are discussed. The IBM 360-390+ series of VMs are described and used as a case study throughout the chapter. The more recently developed VMware hosted virtual machine, which is targeted at IA-32 platforms, is an important second case study.

Applying virtualization to multiprocessor systems is the topic of Chapter 9. Of major interest are techniques that allow the partitioning of resources in large shared memory multiprocessors to form a number of smaller, virtual multiprocessors. These techniques can be implemented in a number of ways, ranging from those re-

lying on microcode support to purely software solutions. Then, ISA emulation in a multiprocessor context is discussed. Although most emulation techniques are the same as with uniprocessors, memory ordering (consistency) constraints pose some new problems. In this chapter IBM logical partitioning (LPAR) and the Stanford Disco research VM are used as case studies.

Finally, Chapter 10, the concluding chapter, looks toward the future and considers a number of evolving VM applications that hold promise. These include support for system security, grid computing, and virtual system portability.

The book also includes an Appendix that reviews the important properties of real machines. The focus is on those aspects of architecture and implementation that are important when virtualization is performed. For some readers this will be familiar material, but other readers may benefit from browsing through the Appendix before proceeding with the book. The Appendix concludes with a brief overview of the IA-32 and PowerPC ISAs, on which many of the examples are based.