An Overview of Virtual Machine Architectures

J. E. Smith and Ravi Nair

Excerpt from "Virtual Machines: Architectures, Implementations and Applications," to be published by Morgan Kaufmann Publishers, 2004.

When early computer systems were being developed, hardware was designed first and software written specifically for that hardware followed. Each system was essentially handcrafted with its own instruction set, and software was developed for that specific instruction set. This included operating system software, assemblers (later compilers), and application programs. With a small user community, relatively simple programs, and fixed hardware, this tightly integrated approach worked quite well, especially while the basic concepts of the stored program computer were still evolving. But user communities grew, operating systems became more complex, and the number of application programs rapidly expanded, so that re-writing and distributing software for each new computer system became a major burden.

The advantages of software compatibility and portability quickly became evident. Furthermore, hardware designers for the various computer companies gravitated toward certain common features. New designs were often similar to previous ones, although usually not identical. It was not until the development of the IBM 360 family in the early 1960s, that the importance of full software compatibility was fully recognized and articulated. 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; the concept of the Instruction Set Architecture (ISA) came into being[1].

In addition, operating systems were developed that could shield application programs from specifics of the hardware, for example, the amount of memory available and the exact characteristics of disk drives. Operating systems were responsible for managing hardware resources, protecting running applications and user data, supporting controlled sharing, and providing other useful functions. Because application programs depend heavily on the services provided by the operating system, these operating system services and the operating system interfaces also became an important point for standardization. Today, there is a relatively small number of standard operating systems. Furthermore, operating systems no longer work in relative isolation;

[1]The term "ISA" is commonly used for distinguishing the formal architecture specification from other less formal uses of the term "architecture" which often include aspects of the hardware implementation.

today they are responsible for controlled protection and sharing of data and resources, extending across networks of computers.

The development of standard interfaces for matching software to underlying hardware has led to a very powerful environment for developing large, complex, and varied software systems. Because of standardized interfaces, software can be made much more flexible and portable than would otherwise be possible. The situation is not perfect, however, because there are a number of different, incompatible ISAs and operating systems. Hence, software portability is limited to those platforms that conform to the same standards as the ones for which the software was developed. User programs are tied to a specific instruction set and operating system. An operating system is tied to a computer that implements a specific instruction set, memory system, and I/O system interfaces. This lack of portability becomes especially restrictive in a world of networked computers where one would like software to move as freely as data.

These limitations on software are only there because the platform, or "machine", on which the software executes is assumed to be a physical entity with certain properties: e.g. it runs a specific ISA and/or operating system. In brief, the software runs on a single type of real machine. *Virtual Machines* (VMs) eliminate this real machine constraint and enable a much higher degree of portability and flexibility. A virtual machine is implemented by adding software to an execution platform to give it the appearance of a different platform, or for that matter, to give the appearance of multiple platforms. A virtual machine may have an operating system, instruction set, or both, that differ from those implemented on the underlying real hardware.

The virtualization of architected elements enables a larger scale virtualization of computer system resources. And in some VM applications, this is the whole point of virtualization. That is, processors, memory, and I/O devices, including network resources can be virtualized. When the state of a process, or system, is virtualized it is essentially separated from a specific piece of physical hardware. This means that a virtual process or system can be temporarily associated with specific physical resources, and that the association can change over time. For example, a server with m processors can support n system virtual machines where $n \geq m$, and the number of physical processors associated with a given virtual machine can change over time as the workloads of the virtual systems change. Similarly, a single physical network interface can serve a number of virtual networks by time-multiplexing the physical resource amongst a number of virtualized network interfaces.

As we shall see, there is a wide variety of virtual machines that provide an equally wide variety of benefits. Virtual machines provide cross-platform compatibility as has already been suggested. They can also be used to increase security, provide enhanced performance, simplify software migration, and enable network-mobile platform-independent software. Over the years, virtual machines have been investigated and built by operating system developers, compiler developers, language designers, and hardware designers. Although each application of virtual machines has its unique characteristics, there also are underlying technologies that are common to a number 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 explore their many applications.

1.1 Standardized Computer System Components: A Two-Edged Sword

Computer systems are constructed of the three major components shown in Figure 0-1: hardware, the operating system, and application programs. The figure illustrates the hierarchical nature of the system and the meshing of its major interfaces. The three major system components are stacked on one another to reflect the direct interaction that takes place. For example, the operating system and application programs interact directly with hardware during normal instruction execution. Because the operating system has special privileges for managing and protecting shared hardware resources, e.g. memory and the I/O system, application programs interact with these resources only indirectly by making operating system calls.

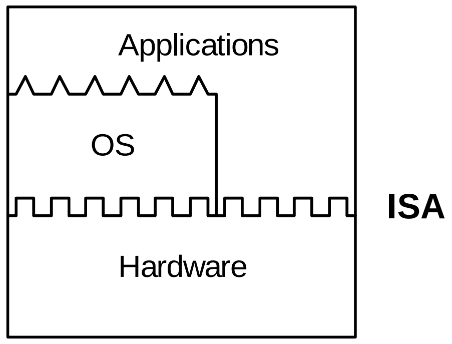


Figure 0-1 A computer system consisting of *Hardware* that implements an *Instruction Set Architecture*, an *Operating System*, and a set of *Application programs.*

Clearly, there are many advantages to the conventional computer system architecture with its well-defined interfaces. Major system design tasks are decoupled; hardware and software designers can work more or less independently. In fact, the three major components are often developed at different companies, and at different times, sometimes years apart. Application developers do not need to be aware of changes inside the OS, and hardware and software can be upgraded according to different schedules. Software can run on different hardware platforms implementing the same ISA, either within the same hardware generation (at different performance levels) or across generations.

Because of its many advantages, the architecture model of Figure 0-1 has persisted for several decades, and huge investments have been made to sustain it. There are also significant disadvantages to this approach, however, and these have become increasingly evident as software and hardware have continued to grow in complexity and the variety of computer applications has broadened.

Problems (or limitations) arise because the major components work together only in the proper combinations. Figure 0-2a shows three popular desktop computer systems, each constructed of hardware, an operating system and application programs. However, the components that form the three systems are not interoperable (Figure 0-2b). Application software compiled for a particular ISA will not run on hardware that implements a different ISA. For example, Apple Macintosh application binaries will not directly execute on an Intel processor. The same is true of system software; Intel PC users had to wait for Windows 3.1 to get a reasonable graphical user interface similar (at least superficially) to the one that Macintosh users had been using for years. Even if the underlying ISA is the same, applications compiled for one operating system will not run if a different operating system is used. For example, applications compiled for Linux and for Windows use different operating system calls, so a Windows application cannot be run directly on a Linux system and vice versa.

In isolated computer systems, lack of software compatibility is bad enough, but in a heavily networked environment the problem becomes even more inhibiting. There are big advantages to viewing a collection of networked computers as a single system ("the network is the computer") where software can freely migrate. This view is obstructed if the networked computers have incompatible ISAs and/or operating systems. That is, if a piece of software is restricted to running on only certain nodes of the network, then a great deal of flexibility and transparency is lost, especially when the network is as large and varied as the internet.

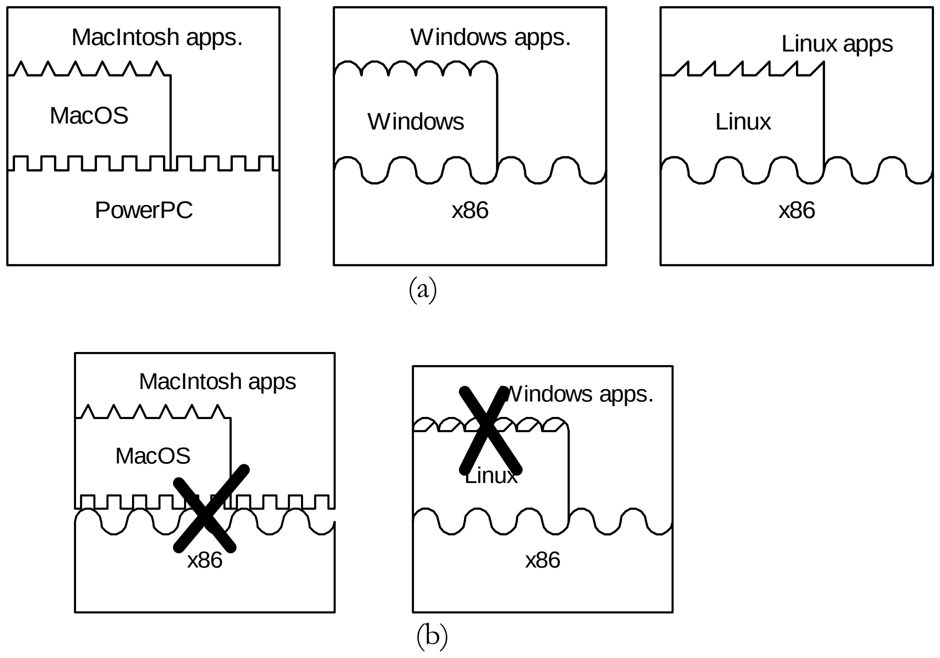


Figure 0-2 Interoperability
a) Three popular computer systems composed of different ISAs, OSes, and Applications.
b) Although highly modular, the major system components are not always interoperable.

A second problem is that innovation is sometimes limited by the need to support interfaces developed years, possibly decades earlier. For example, implementing new software concepts may be inhibited by an old "legacy" ISA. Hardware development may also be inhibited by the need to support ISA features that restrict high performance innovations. And, even when ISAs are extended to allow innovation, as they sometimes are, they must continue supporting old ISA features that are seldom used and become excess baggage.

A third problem is that optimization across major interfaces is difficult. The interfaces allow independent development of the major components, but this can also be problematic. The developers on each side of an interface seldom communicate (often they are working at different companies and at different times), so it is very difficult to cooperate closely on optimizations that cross an interface. For example, the separate development of application software and hardware means that program binaries are often not optimized for the specific hardware on which they are run. Typically only one version of a binary is distributed, and it is likely optimized for only one processor model (if it is optimized at all). Furthermore, compiler development usually lags processor development so binaries are often compiled for an earlier generation processor than on which they are being run.

Finally, in a traditional computer system, one operating system is matched with one hardware platform and all applications co-exist under the management and protection of the single operating system. Not only does this constrain all the system users to the same OS, but it also opens opportunities for exploiting security "holes" in the operating system. That is, the degree of isolation among the application programs is limited by the shared system software. This may be especially important when a large hardware system, e.g. a server, is to be shared by different groups of users who would like to be assured of a secure environment.

1.2 Virtual Machine Basics

The above problems can be solved by implementing a layer of software that provides a virtual machine environment for executing software. One type of virtual machine (VM) is illustrated in Figure 0-3 where virtualizing software is placed between the underlying machine and conventional software. In this example, virtualizing software translates the hardware ISA so that conventional software "sees" a different ISA from the one supported by hardware. As we shall see, virtualizing at the ISA level is only one possibility, but it is adequate for illustrating the range of VM applications. The virtualization process involves 1) the mapping of virtual resources, e.g. registers and memory, to real hardware resources and 2) using real machine instructions to carry out the actions specified by the virtual machine instructions, e.g. emulating the virtual machine ISA.

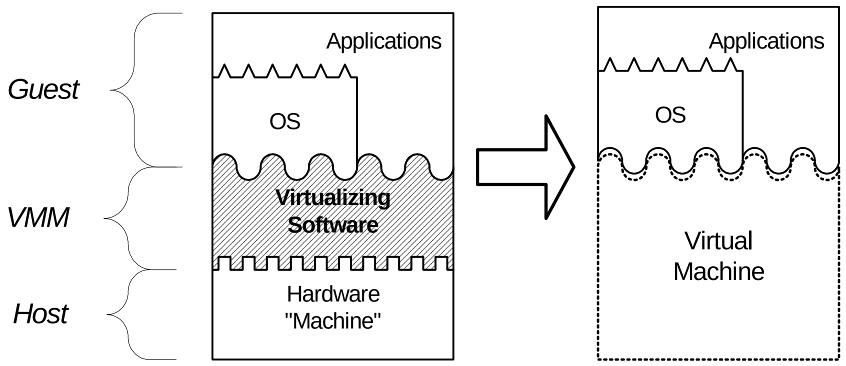


Figure 0-3 Virtualizing software can translate the ISA used by one hardware platform to another, forming a Virtual Machine, capable of executing software developed for a different set of hardware.

With regard to terminology (see Figure 0-3), we usually refer to the underlying platform as the "host", and the software that runs in the VM environment as the "guest". The virtualizing software will often be referred to as the "virtual machine monitor" (VMM), in accordance with the term used in the original VMs developed in the late 1960's. Occasionally, we will use other more specific names for the virtualizing software, depending on the type of VM that is being discussed.

Virtualizing software can be applied in several ways to connect and adapt the three major system components (see Figure 0-4). As just illustrated, *emulation* (Figure 0-4a) adds considerable flexibility by permitting "mix and match" cross-platform software portability. Virtualizing software can enhance emulation with *optimization,* by taking implementation-specific information into consideration as it performs emulation, or it can perform optimization alone, without emulation (Figure 0-4b). Virtualizing software can also provide resource *replication,* for example by giving a single hardware platform the appearance of multiple platforms (Figure 0-4c), each capable of running a complete operating system and/or a set of applications. Finally, the various types of virtual machines can be composed (Figure 0-4d) to form wide variety of architectures, freed of many of the traditional compatibility constraints.

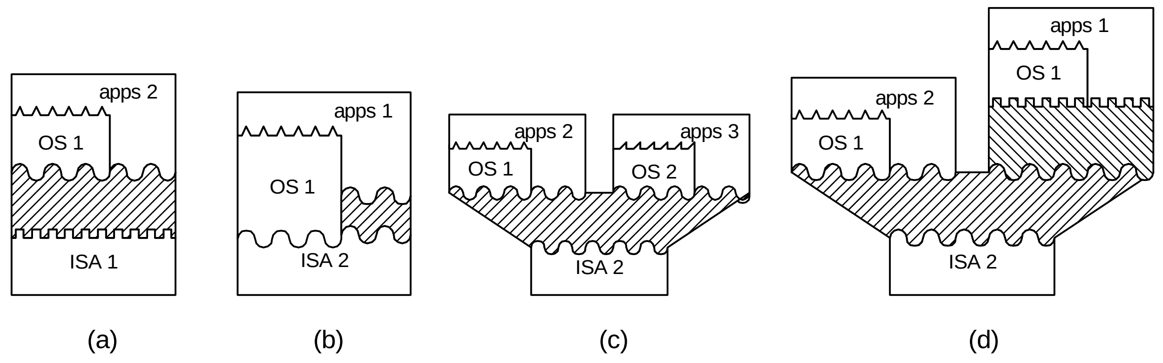


Figure 0-4. Examples of virtual machine applications.

a) emulating one instruction set with another
b) optimizing an existing application binary for the same instruction set,
c) replicating a virtual machine so that multiple OSes can be supported simultaneously,
d) composing virtual machine software to form a more complex, flexible system.

Given the wide variety of applications, VM technologies are widely used today to allow interoperability of the major system components. Furthermore, because of the heavy reliance on a few standards and consolidation in the computer industry, it seems likely that any major innovation, e.g. a new ISA, new OS, or new programming language will leverage VM technology. Consequently, for constructing modern systems, virtualizing software has essentially become a fourth major system component that merits equal standing with hardware, operating systems and application software.

In computer systems, with their many levels of abstraction, the meaning of "machine" is a matter of perspective. From the perspective of a *process*, the machine consists of a 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 system, as perceived by the process, is rather abstract. Disks and other secondary storage appear as a collection of files to which the process has access permissions. In the desktop environment, the process can interact with a user through a window (or windows) that it creates within a larger graphical user interface (GUI). The only way the process can interact with the I/O system of its machine is via operating system calls, either directly, or through libraries that are supplied to 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.

From a higher level perspective, an entire *system* is supported by an underling 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 physical 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.

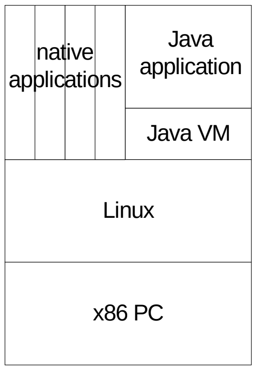


Figure 0-5. A process supported by a guest Java Virtual Machine executes alongside native processes on a Linux/x86 host platform.

Just as there is a process perspective and a system perspective of what a machine is, one can also provide virtual machines at either the process level or the system level. As the name suggests, a *process virtual machine* is capable of supporting an individual process. A process virtual machine is created along with its guest process and terminates when the guest process terminates. For example, consider a host platform that consists of Linux running on Intel x86 PC hardware (Figure 0-5). This system clearly can support native x86 applications that are compiled for a Linux system. However, this system can also act as a host for supporting guest Java processes via Java virtual machines. The Java VM environment can execute Java bytecode programs and perform I/O operations through Java libraries. The same system could also support PowerPC/Linux guest applications via process level virtual machines that are capable of emulating the PowerPC instructions.. All these processes can reside simultaneously within the same system environment, and they may interact in the ways that processes sharing a system normally do for example, a PowerPC process could create an x86 process and communicate with it via shared memory. In a desktop system, for example, they would each have individual windows, all appearing within the same X-windows GUI.

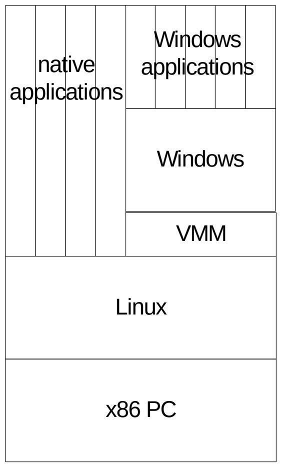
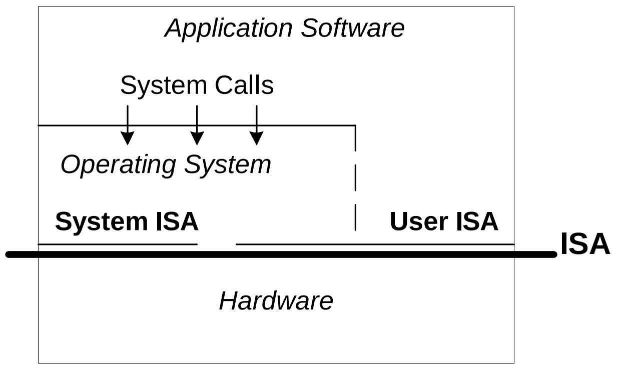


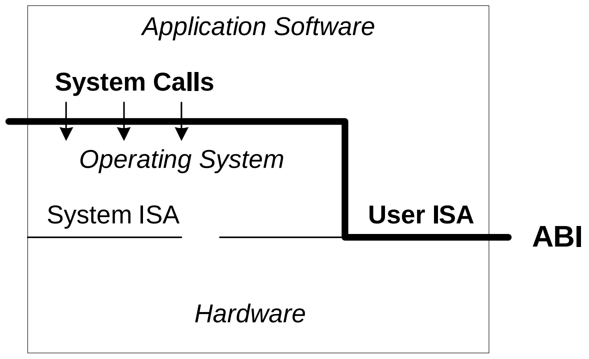
Figure 0-6. A Windows system supported as guest virtual machine running on a Linux/x86 host platform.

A *system virtual machine* provides a complete system environment. This environment can support multiple user processes, includes a file system, provides the processes with access to I/O devices, and, on the desktop, it supports a GUI. Again, consider a platform that consists of Linux running on x86 hardware (Figure

0-6). The native system is Linux/x86, but the platform can also serve as a host for supporting a virtual Windows system, accessed through the Windows GUI, that contains a (virtual) Windows file system and can run Window applications. The Windows processes interact with each other, just as on a real windows system. And, they can interact with the native UNIX system via a network interface, just as if it were a physically separate system. However, the UNIX and Windows processes do not interact directly as if they were part of the same system (because they are not part of the same system); for example, a UNIX process cannot spawn a Windows process.



(a)



(b)

Figure 0-7. System Interfaces
a) Instruction Set Architecture (ISA) interface
b) Application Binary Interface (ABI)

For implementing virtual machines, there are two standardized interfaces that will be of interest to us. The interfaces roughly correspond to the process level and the system level. These interfaces are shown in more detail in Figure 0-7. The ISA, shown in Figure 0-7a, includes both *user* and *system* instructions. The user instructions are available to both application programs and to the operating system. The system instructions include *privileged* operations that permit the direct manipulation, allocation, and observation of shared hardware resources (the processor, memory and I/O).

Figure 0-7b illustrates the Application Binary Interface (ABI). The ABI has two major components. The first is the set of all user instructions; system instructions are not included in the ABI. At the ABI level, all application programs interact with the shared hardware resources indirectly, by calling the operating system via a *system call* interface, which is the second major part 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 a system call instruction that transfers control to the operating system in a manner somewhat similar to a subroutine call, except the call target address is forced to be a specific address in the operating system. Arguments for the system call are passed through registers and/or a stack held in memory, following specific conventions that are part of the system call interface.

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.

1.3 Process VMs

Process level VMs provide user applications with a virtual ABI environment. In their various forms, process VMs can provide replication, emulation, and optimization. The following subsections describe each of these. Note that for many of VMs, it is common practice to use some term other than "Virtual Machine" to describe the function being performed. Nevertheless, they are for all intents and purposes virtual machines – in spirit if not in name.

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. Conventional multiprogramming provides each user process with 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 OS provides replicated process level virtual machines for each of the concurrently executing applications. And, there is virtually an unlimited number of such processes – the process virtual machine can be freely replicated.

1.3.2 Emulation and Dynamic Binary Translators

A more challenging problem for process level virtual machines is to support program binaries compiled to a different instruction set that the one executed by the underlying hardware. Such an emulating process virtual machine is illustrated in Figure 0-8. Application programs are compiled for a *source ISA*, but the hardware implements a different *target ISA*. As shown in the figure, the operating system is the same for both the guest process and the host platform. The example illustrates the Digital FX!32 system. The FX!32 system could run Intel x86 application binaries compiled for Windows NT, on an Alpha hardware platform also running Windows NT.

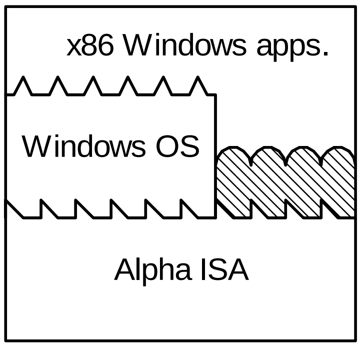


Figure 0-8. A process VM that emulates guest applications. The Digital FX!32 system allows Windows x86 applications to be run on an Alpha Windows platform.

Because of the different ISAs, it is necessary for the virtual machine to emulate execution of the source ISA. The most straightforward emulation method is interpretation. An interpreter program executing the native, target ISA fetches, decodes and emulates execution of individual source instructions. This can be a relatively slow process, requiring on the order of tens of native instructions for each source instruction interpreted.

For better performance, binary translation is typically used. With binary translation, blocks of source instructions are converted to native 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 commonly used with this type of process virtual machine, they are often 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 model combined with profiling. 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 occur over the course of a single program run. In the case of FX!32 mentioned above, translation occurs incrementally between program runs.

1.3.3 Dynamic Optimizers

Most dynamic translators not only translate from source to target code, but they also perform some optimizations in the process. This leads naturally to virtual machines where the instruction sets used by host and the guest are the same, and optimization is the sole purpose of the virtual machine. Dynamic translators are implemented in a manner very similar to emulating virtual machines, including staged optimization and software caching of optimized code. An example of such a *dynamic optimizer* is the Dynamo system, developed as a research project at Hewlett-Packard. A dynamic optimizer can collect statistics on a running program, i.e. construct a *profile*, and then use this profile information to optimize the binary on-the-fly.

1.3.4 High Level VMs: Complete Platform Independence

For the VMs described above, cross-platform portability is clearly a very important objective. For example, the FX!32 virtual machine enabled portability of application software compiled for a popular platform (x86 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 effort. For example, if one wanted to run x86 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. Furthermore, the problem would be even more difficult, if not impossible, if the host platforms run a different OS than the one for which the binary is compiled.

Full cross-platform portability is more easily achieved by taking a step back and designing it into programs in the first place. One way of accomplishing this is to *design* a process level VM at the same time as an application development environment, including a high level language (HLL), is being defined. Here, the VM environment does not correspond to a particular hardware platform. Rather it is designed for portability and to match the features of the HLL used for application program development. These High Level VMs are similar to the ISA-specific process VMs described above. They focus on supporting applications and they tend to minimize hardware related-features (and OS-specific features) because these would compromise platform independence.

This type of virtual machine first became popular with the Pascal programming environment. In a conventional system, Figure 0-9a, 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 an object file containing machine code for a specific ISA and OS. This object file is then 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.

With Pascal, this model was changed (Figure 0-9b). The steps are similar to the conventional ones, but the point at which distribution takes place is at a higher level. In Figure 0-9b, a conventional compiler front-end generates abstract machine code, which is very similar to an intermediate form. In Pascal, this "P-code" is for a rather generic stack-based ISA. P-code is in essence the machine code for a virtual machine. It is the P-code that is distributed to be executed on different platforms. For each platform, a VM capable of executing the P-code is implemented. In its simplest form, the VM contains an interpreter that takes each P-code instruction, decodes it, and then performs the required operation on state (memory and the stack) that is used by the P-code. 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 re-compiled (binary translated) into machine code for the host platform that can be directly executed.

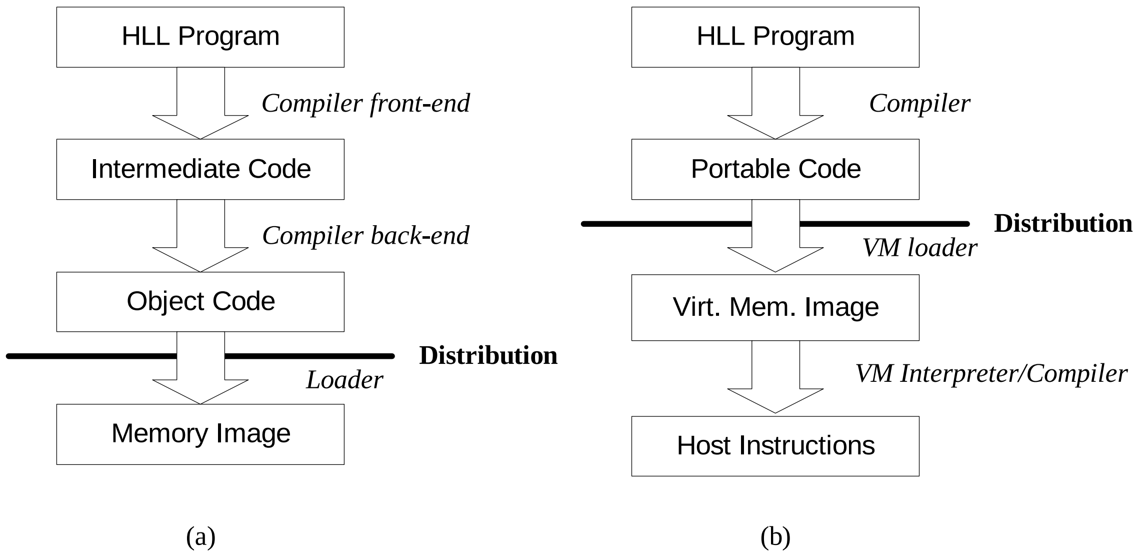


Figure 0-9. High Level Language environments
a) A conventional system where platform-dependent object code is distributed
b) A HLL VM environment where portable intermediate code is "executed" by a platform-dependent virtual machine.

An advantage of a high level VM is that software can be completely portable, provided the VM can be implemented on a target platform. While this would take some effort, it is a much simpler task than developing a compiler for each platform and re-compiling an application beginning with the HLL whenever it is to be ported. It is also much simpler than developing a conventional emulating process VM for a typical real-world ISA.

The Java VM architecture is a more recent, and a widely used example of a high level VM. Platform independence and high security are central to the Java VM architecture. The ISA chosen for Java definition is referred to as the "Java Bytecode" instruction set. Like P-code, this instruction set is stack-based (to eliminate register requirements) and has a very general bit width-independent data specification and memory model. In fact, the memory size is essentially unbounded, with garbage collection as an assumed part of the implementation. Because all hardware platforms are potential targets for executing Java programs on a Java virtual machine, application programs are not compiled for a specific OS. Rather, just as with P-code, there is a set of standard libraries whose calling interface and functions are parts of the Java virtual machine architecture.

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 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 computer hardware 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 several users. Figure 0-10 illustrates these classical system VMs.

For system VMs, replication is the major feature provided by a VMM. The central problem is 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 a system ISA operation involving 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. Multiple guest OSes can be supported; in Figure 0-10 there are two.

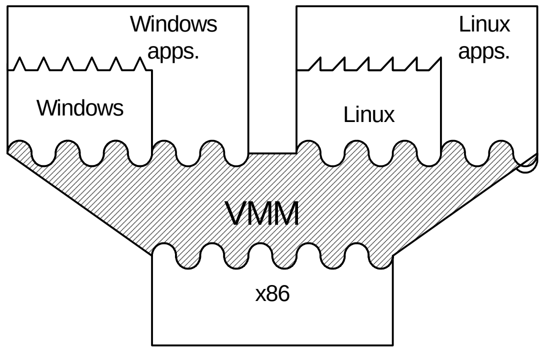


Figure 0-10. An example system VM -- supporting multiple OS environments on the same hardware.

This type of replicated VM system has a number of advantages. The first, and probably most important today is that multiple OS environments can be simultaneously supported, so applications developed for different OSes can be simultaneously executed on the same hardware. For example, DOS, Windows NT, OS/2, Linux, and Solaris operating systems are all available (or have been) for the x86-based PC platform. And, different application programs have been developed for each of the OSes. This poses a problem if a user wants to run an application written for Windows NT on a PC that is currently running Solaris. There is a similar problem if multiple users are sharing a larger system, say a server, and different users prefer (or require) different operating systems.

Another important application, and one that could possibly be the most important in the future, is that the separate virtual environments provide a security "firewall" for protecting independent groups of users and applications. Also, OS software development can be supported simultaneously with production use of a system. This replicated VM approach has been very successful for IBM; it has continued to evolve and is a key part of large multiprocessor servers -- today's equivalent to the mainframes of the '60s and'70s.

1.4.1 Implementations of System VMs

From the user perspective, most system VMs provide more-or-less the same functionality (although performance may vary widely). 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 there is a number of choices for where the VMM can be placed. Summaries of two of the more important implementations follow.

In Figure 0-10, the "classic" VM, the VMM is placed on "bare" hardware, and virtual machines fit on top. The VMM runs in the most highly privileged mode, while all the guests run with lesser privileges. Then, the VMM can intercept and implement all the guest OS's actions that interact with hardware resources -- *in a completely transparent way*. In many respects, this system VM implementation 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, installing the VMM, then installing guests on top. Another disadvantage is that 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 done 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, a modern VM that runs on x86 hardware platforms.

1.4.2 Whole System VMs: Emulation

In the "classic" system VMs described above, all the operating systems (both guest and host) and applications 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.

This situation leads to an emulating system VM, 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 implementaion method is to place the VMM and guest software on top of a conventional host OS running on the hardware.

Figure 0-11 illustrates a whole system VM built on top of a conventional system with its own OS and application programs. 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.

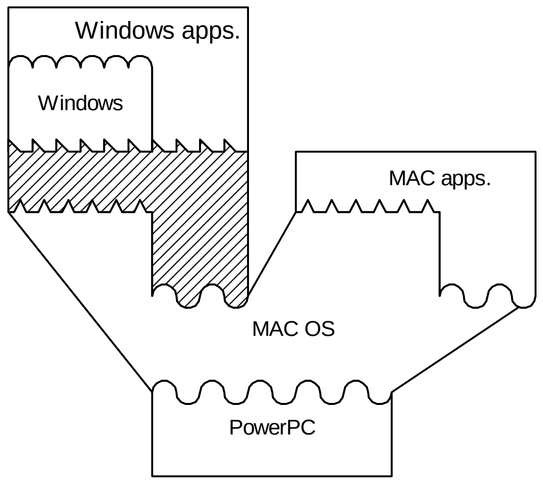


Figure 0-11. A whole system VM that supports a guest OS and applications, in addition to host applications.

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 to the host OS. Even if binary translation is used, it is tightly constrained because translated code cannot easily 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 difficult because the VMM has no direct access to hardware resources and can only go through the host OS via the system calls that it provides.

1.4.3 Co-Designed VMs: Optimization

In all the VM models discussed thus far, the goal has been flexibility 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. beyond native hardware 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 new ISAs and/or hardware implementations for improved performance, power efficiency, or both. Co-designed VMs may be based on hardware that uses a non-standard, proprietary ISA. In fact, the native ISA is developed along with the hardware implementation to enhance and enable better hardware implementation features. This native ISA may be completely concealed from all conventional software that runs on the host processor.

In general, the host 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 (Figure 0-12), there are no native ISA applications (although one can imagine situations where they may eventually be developed). It is as if the VM software is, in fact, part of the hardware implementation. Because the goal is to provide a VM platform that looks exactly like a native hardware platform, the software portion of the VM uses a region of memory that is not visible to any application and system software.

This concealed memory is carved out of main 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 control 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 often coupled with code optimization. Optimization can be performed at the translation time, and/or it can be performed as an ongoing process as a program runs, depending on which code sequences are more frequently executed.

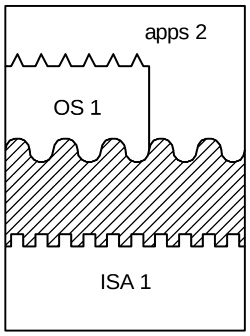


Figure 0-12. Co-Designed VM -- VM software translates and executes code for a concealed native ISA.

The two best examples of co-designed VMs are the IBM Daisy processor and the Transmeta Crusoe. In both these processors, the underlying hardware happens to use a native VLIW instruction set; the guest ISA for Daisy is PowerPC and for Transmeta it is the Intel x86. The objective of the Daisy project is to provide the simplicity and performance advantages of VLIW while executing existing software. The Transmeta designers focused on power saving advantages of simpler VLIW hardware.

Besides code translation and optimization, VM software can also be used in other ways to optimize the usage of hardware resources. For example, the VMM can read and analyze performance-monitoring data collected in the hardware. It can then reconfigure resources, e.g. a cache memory, for better performance or power savings. If a fault is detected by hardware, the hidden VMM may be able to isolate the fault and reconfigure hardware to avoid the faulty portion. In effect, the VMM acts as a small operating system for native processor resources in a manner similar to the way the conventional operating system manages system resources.

1.5 Virtualization

The computational function carried out by a computer system is specified in terms of architected state (registers, memory) and instructions that cause changes in the architected state. In the physical implementation, there is also state in the form of latches, flip-flops, and DRAM, for example. And, at any given time, there is a mapping from the architected state to the implementation state (Figure 0-13). Instructions in the architecture cause state transitions in the implementation, such that the change in implementation state can be interpreted as a corresponding change in the implementation state. Over the course of a computation, the exact implementation element (e.g. a register) that corresponds to a particular architected element may change; for example when register renaming is used. Nevertheless, at certain points during the computation, it must be possible to construct the "precise" architected state. In a modern hardware implementation, one of the more challenging situations is when an exception condition (trap or interrupt) occurs, and the entire architected state must be materialized.

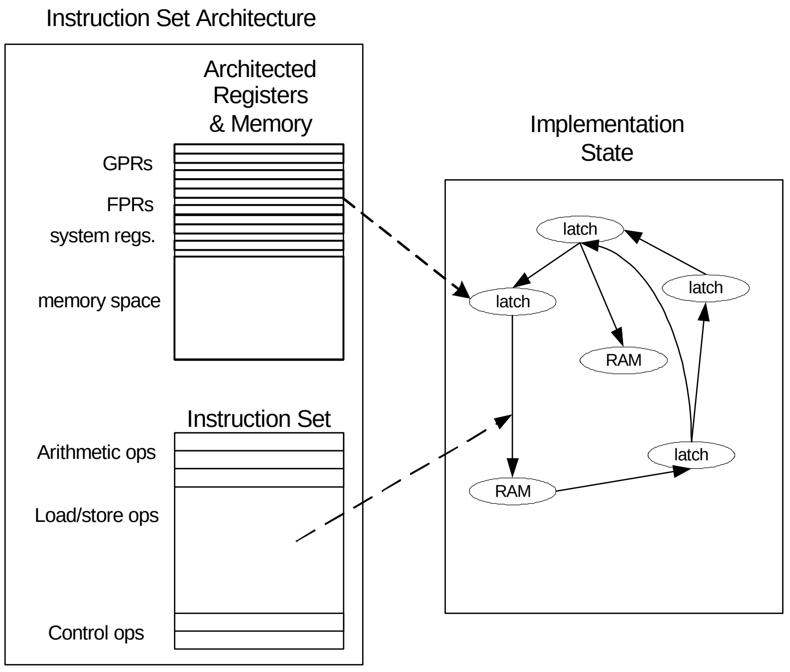


Figure 0-13. In a computer system, architected state is mapped to implementation state. Instruction execution causes state transitions in the implementation.

In early machines the architected and implementation state were more-or-less equivalent, that is, a register in the physical design would typically be visible as a register in the instruction set architecture. Today, there is often more implementation state than architected state. This implementation state includes pipeline latches, branch predictor contents, rename registers, etc. For good performance, the architected state is usually held in an implementation equivalent, i.e. an architected register is held in a physical register. However, as far as *function* is concerned, state is state, no matter how efficient or inefficient it is to access. For example, architected register state could be held in implementation DRAM.

When a computational function is *virtualized*, there is an analogous situation to the one just described (see Figure 0-14). In a virtual machine, the architected guest state maps to architected host state and the guest instructions are emulated by host instructions that cause corresponding state transitions. In order to get high performance, this architectural faithfulness at the instruction level is often achieved by maintaining architectural faithfulness at some higher level of granularity, while providing mechanisms to backtrack and recover the precise state at some particular instruction whenever required. The inherent assumption is that events that need such recovery of precise state are infrequent, so that, for large stretches of time, the machine can be running at full speed (or nearly so).

A practical difference between the virtual machine implementation and a real machine implementation is that the real machine is implemented with a specific instruction set architecture in mind. Consequently, there is usually a good match between the architected and implementation state, and the hardware that causes implementation state transitions is well-matched for the architected instructions. In the virtual machine case, however, the guest and host are often specified independently of each other (an exception is co-designed VMs). This means that state mapping and instruction mapping may not necessarily be efficient. For example, the host may have fewer registers than the guest. The efficiency of virtualization depends on the type of VM being implemented. In the remainder of this section, we will discuss the virtualization process, i.e. the mapping of guest onto host, for each of the VMs, and will evaluate virtualization efficiency in general terms. This discussion will provide an overview of the major issues to be discussed throughout the book as we describe VM implementations.

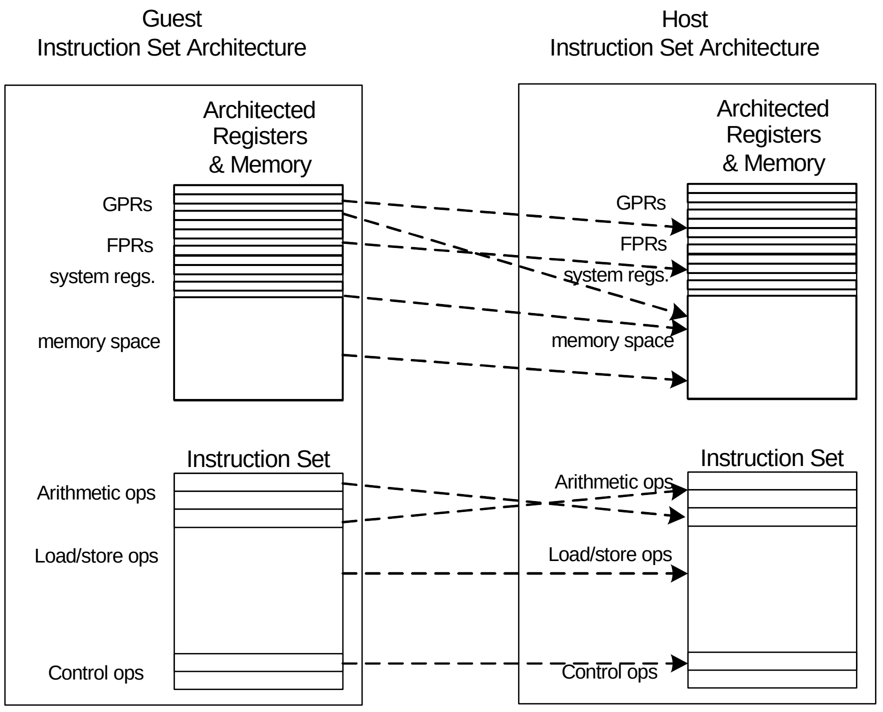


Figure 0-14. In a virtual machine, guest state is mapped to host state.

There are three aspects to the virtualization process that we will focus on: mapping of state (including both register and memory state), instruction mapping (emulation), and precise state materialization when exceptions occur. While not all-inclusive, these aspects are the probably the more important ones and collectively they represent the issues that must be considered when a process or system is virtualized.

1.5.1 Process VMs

As described in Section 1.3, there is a variety of process level virtual machines, i.e. those VMs where the process views a virtual execution environment consisting primarily of user level instructions and operating system calls. For each of the process VMs discussed, we will briefly summarize the major issues related to the virtualization process: state mapping, instruction mapping, and materialization of precise state.

The easiest process virtual machine to implement is multiprogramming – because virtually all computer systems today are designed to support a multiprogrammed environment. First, the state of the process, both registers and memory map naturally in a 1 to 1 fashion with the underlying architected state. Instructions are natively executed, that is, they require no emulation or any other special handling. Finally, the state can be captured precisely whenever an exception occurs, and this feature is provided by the hardware implementation. Table 0-1 summarizes the key aspects of the virtualizing process virtual machines; multiprogramming is in the first row of the table.

Table 0-1. Key aspects of the virtualizing process VMs

	State		**Instructions**	**Preciseness**
	Registers	**Memory**		
Multiprogramming	Mapped 1:1	Mapped 1:1	native	provided by hardware
Dynamic Translation	Mapped to host regs. as available	Mapped to available memory	emulated	provided by VM software
Dynamic Optimization	Mapped 1:1	Mapped 1:1	block-level translated	provided by VM software
HLL VMs	Mapped to host regs. as available	Mapped to available memory	emulated/ just-in-time compiled	provided by VM software

Process virtual machines can also provide dynamic translation for cross-platform compatibility. For these VMs (second row of Table 0-1) the guest state is mapped to the host state "as available". In particular, all guest registers can be mapped to host registers, *provided there are enough host registers.* In cases where there are not, e.g. if the PowerPC (32 general registers) is being virtualized on an x86 (8 general registers), then at any given time, some of the guest registers will have to be mapped to memory locations, which will likely reduce efficiency. Likewise, the memory space of the guest is mapped into the memory space of the host. If the guest address space is larger, then the VMM must manage the situation, for example by keeping some of the guest state on disk. Because the guest and host instruction sets are different, emulation of some type must be used. The type of emulation depends on performance requirements and can be either interpretation, translation, or both. Precise state is also provided by emulation software; that is, the software must keep track of the correct guest state and must be able to materialize the state when an exception occurs.

With dynamic optimizers, the guest and host architectures are the same, as in multiprogramming, so ostensibly the register and memory state are relatively easy to map in a 1 to 1 fashion. However, because optimizations often increase register pressure, this mapping may at times be modified with some register state being held temporarily in memory. Although the instruction sets of the host and the guest are the same, many of the same techniques that are used for emulation are also used for dynamic optimization. In particular, blocks of code are scanned and optimized, with the optimized versions being cached for multiple re-use. Because many dynamic optimizations systems are transparent to the user, exceptions must result in materialization of a precise state with respect to the original (unoptimized) code. Consequently, the VM software will typically track and recover correct guest state, just as it does when dynamic translation is performed.

Finally, high level language virtual machines are in many respects similar to dynamic translating VMs. An important difference, however, is that HLL VMs are usually designed with portability in mind. Consequently, although the virtualization issues may be similar to the issues with dynamic translators, the solutions are often more straightforward. For example, some HLL VMs use a stack to specify instruction operands rather than registers; hence, there is no problem with register mapping, and whatever registers happen to be available in the host can be used for optimizations. The memory model in HLL VMs is usually more abstract than in a conventional ISA, so, once again, mapping of state is simplified. Regarding instruction mapping, HLL VMs are usually designed with ease of interpretation in mind. Consequently, they usually have a rather "vanilla" flavor, and they are often designed to be compiled (e.g. translated) at a procedure (or method) level. Often compilation is performed "just-in-time", i.e. it is deferred until a method or procedure is invoked for the first time. Finally, precise state must be provided by the VM software, but the process state and requirements for exception handling are often specified in ways that simplify the problem.

1.5.2 System VMs

In system VMs, both user code and system code runs on a virtual machine, and the virtualization techniques may depend on whether code user or privileged system code is being run. Consequently when we summarize the major aspects to virtualization, we distinguish these two types of code. Table 0-2 summarizes the important considerations when implementing system level VMs.

Table 0-2. Key aspects of virtualizing system VMs

		State		Instructions	Preciseness
		Registers	**Memory**		
"Classic" System VMs	**User**	Mapped 1:1	Mapped 1:1	native	provided by hardware
	System	In memory	Mapped 1:1	Native/some emulated	provided by hardware
Whole System VMs	**User**	Mapped to available host regs.	Mapped to available memory	emulated	provided by VM software
	System	In memory	Mapped to available memory	emulated	provided by VM software
Co-Designed VMs	**User**	Mapped 1:1	Mapped 1:1	block-level translated	provided by hardware/VM software
	System	Mapped 1:1	Mapped 1:1	block-level translated	provided by hardware/VM software

Classic system VMs run guest software that is developed for hardware running the same ISA as the host platform, hence, in user mode, the registers and memory state can be mapped 1 to 1, and in most VM implementations, the instructions can be natively executed. There may be exceptions to this, however, depending on properties of the instruction set that will be described in Chapter 8. The system code requires special handling because in the virtual machine implementation, this code does not actually run in a privileged mode. Only VMM software runs in system mode because the VMM keeps overall control of the system resources and allocates them among the various guest virtual machines. Consequently, the system registers (those used for managing hardware resources and accessible only by system instructions) are held in memory and are maintained by the VMM. The real system registers are used by the VMM for managing the hardware resources.

Whole system VMs have many of the characteristics of the process VMs that use dynamic translation. The major difference is that both system and user instructions must be emulated. That is, it is the ISA interface that is of interest rather than the ABI interface. Both user registers and memory state of the guest are mapped to guest state "as available". Any excess guest register state is mapped to memory. The system registers, i.e. those used for managing hardware resources are mapped to memory and are maintained by the VMM. In many whole system VMs, the VMM software also runs in user mode and relies on an underlying host operating system for managing memory resources.

Co-designed VMs make an interesting case because here the hardware is designed specifically to implement a guest's instruction set. Consequently, there will most likely be adequate host register and memory resources for all of the guest state to be mapped 1 to 1. Instructions are primarily emulated via dynamic binary translation and are optimized for high performance. Because hardware is specially designed, it may be able to support precise state recovery for exceptions, or it may do it cooperatively with VM software. Finally, because there is typically only one guest in a co-designed VM, and the guest can be provided with both a system and user mode of operation, even the system registers can be maintained in hardware registers.

1.6 Summary and a Taxonomy

We have just described rather broad array of VMs, with different goals and different implementations. To put them in perspective and organize the common implementation issues, following is an overall taxonomy. First, VMs are divided into the two major types: Process VMs and System VMs. In the first type, the VM supports the ABI – user instructions plus system/library calls, in the second, the VM supports a complete ISA – both user and system instructions. The remainder of the taxonomy is based on whether the guest and host use the same ISA. Using this taxonomy, Figure 0-15 shows the "space" of virtual machines that we have identified.

On the left side of the figure are process VMs. There are two types of process VMs where the host and guest instruction sets are the same. The first is multiprogrammed systems, where virtualization is a natural part of multiprogramming and is supported on most of today's systems. The second is dynamic optimizers, which transform guest instructions only by optimizing them, and then execute them natively. The two types of process VMs that do provide emulation 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.

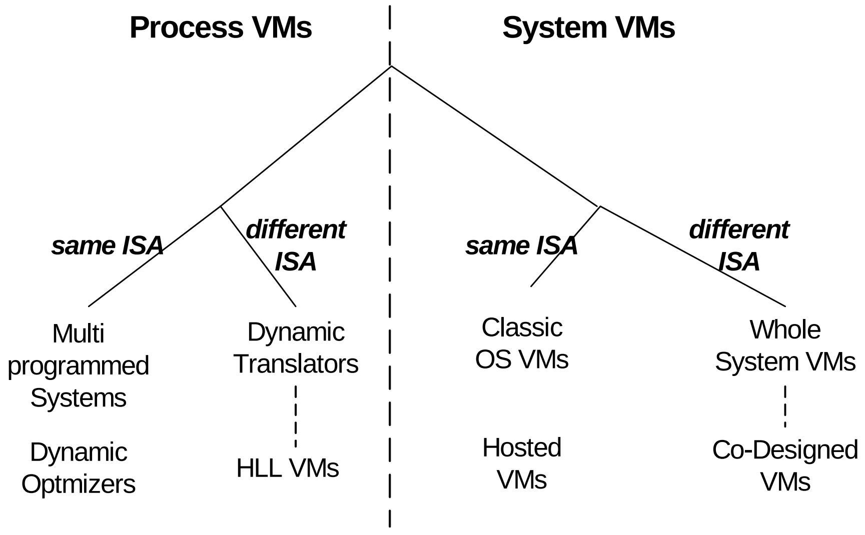


Figure 0-15. A taxonomy of virtual machine architectures.

On the right side of the figure are system VMs. These range from Classic OS VMs and Hosted VMs, where replication – and providing isolated system environments – is the goal, to Whole System VMs and Co-Designed VMs where emulation is the goal. With Whole System VMs, performance is often secondary, in favor of accurate functionality, while with Co-Designed VMs, performance (or power efficiency) is the major goal. Here, Co-Designed VMs are "dotted line" connected because their interface is at a lower level than other system VMs.

THE POWER OF VIRTUAL MACHINES

A good way to end this summary is with an example of a realistic system that could conceivably be in use today (Figure 0-16). The example clearly illustrates power 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 x86/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 x86 hardware is in fact a Transmeta Crusoe, a co-designed VM implementing a VLIW ISA with binary translation to support x86. Through the magic of VMs, a Java bytecode program is actually executing as native VLIW.

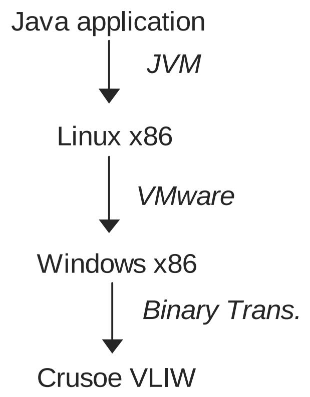


Figure 0-16. Three levels of VMs: a Java application running on a Java VM, running on an OS VM, running on a co-designed VM.