

An Overview of Virtual Machine Architectures

J. E. Smith

October 27, 2001

1 Introduction

When early computer systems were being developed, hardware was designed first, and machine-level software 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 and relatively simple programs, this whole-system 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 became self-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 recognized. The IBM 360 series had a number of models covering a broad spectrum of price and performance levels -- yet they were all designed to run exactly 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 being¹.

In addition to developing hardware, computer companies also developed operating systems to manage hardware resources, protect running applications and user data, support controlled sharing, and provide other useful functions. Because application programs depend heavily on the

¹The 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.

services provided by the operating system, these operating system services and the operating system interfaces also became standardized.

Today, virtually all computer systems are composed of the three major components shown in Fig. 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.

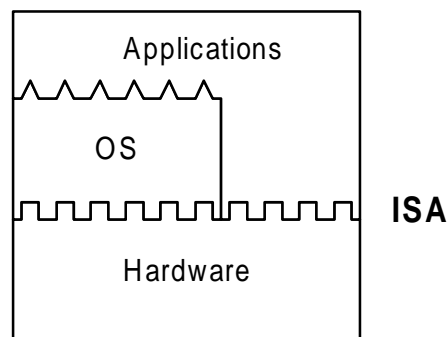


Fig. 1. A computer system consisting of Hardware that implements an Instruction Set Architecture, an Operating System, and a set of Application programs

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. 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 Fig. 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.

One fundamental problem is illustrated in Fig. 2. The major components only work in the proper combinations. Fig. 2a shows three popular computer systems, each constructed of hardware, operating system and application programs. However, the components that form the three systems are not interoperable (Fig. 2b).

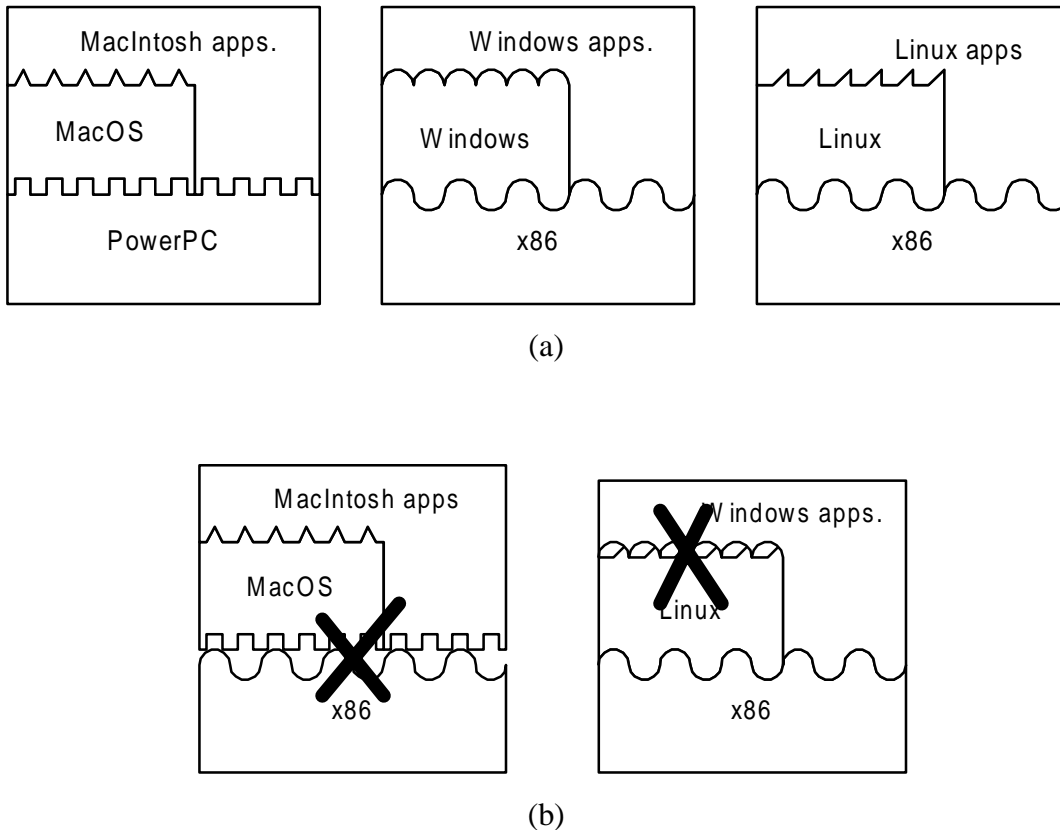


Fig. 2 a) Three popular computer systems composed of different ISAs, OSeS, and Applications.
b) Although highly modular, the major system components are not intero-perable.

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 3.0 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 system calls, so a Windows application cannot be run directly on a Linux system and vice versa.

A second fundamental problem is that innovation is inhibited 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 fundamental problem is that optimization across the major interfaces is difficult. The interfaces allow independent development of the major components, but this is a two-edged sword. 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.

As another example, the independent development of the operating system and hardware means physical implementation features are hidden from software. This is often a positive feature. However, in some cases such as power management and advanced performance features, it may be useful for system software to monitor and adjust the underlying hardware configuration. This requires a high degree of cooperation between hardware implementers and system software developers, and some hardware implementation features must be incorporated into the system software.

To solve the above problems, special "coupling" software can be used to connect the major components. For example, in Fig. 3, coupling software is placed between the hardware (the "machine") and software. This software translates the ISA so that conventional software "sees" one ISA while the hardware supports another. In effect, the hardware plus translating software presents a *Virtual Machine* (VM) to the conventional software. This is in contrast to the underlying

hardware that is a real machine. Consequently, we refer to all such coupling software as “virtual machine software”.

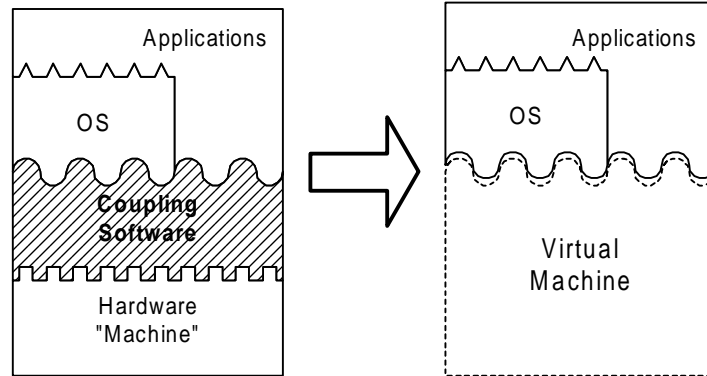


Fig. 3. “Coupling Software” can translate the ISA used by one hardware platform to another, forming a Virtual Machine, capable of executing software developed for a different set of hardware.

Virtual Machine software can be applied in several ways to connect and adapt the three major system components (see Fig. 4). As just illustrated, VM software *translation* (Fig. 4a) adds considerable flexibility by permitting “mix and match” cross-platform software portability. VM software can also enhance translation with *optimization*, by taking implementation-specific information into consideration as it performs the translation, or it can perform optimization alone without translation (Fig. 4b).

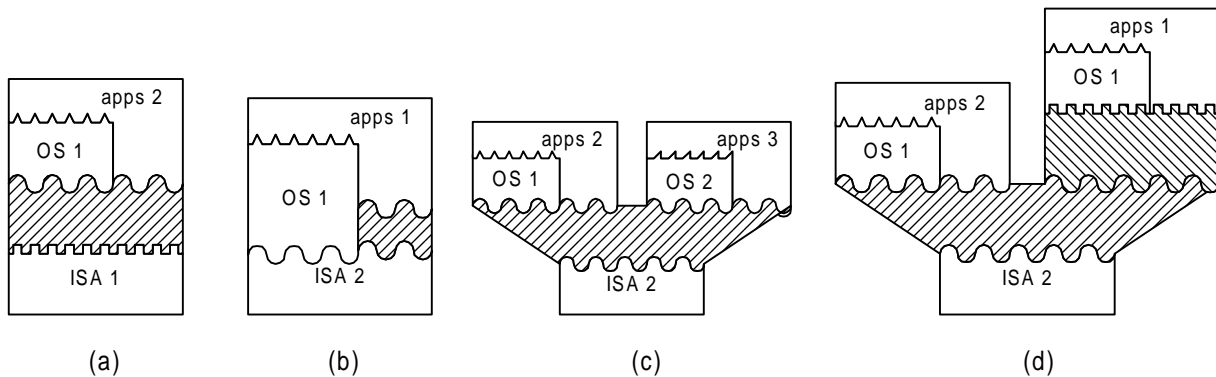


Fig. 4. Examples of virtual machine software being used to connect the major system components.

- a) translating from one instruction set to another
- b) optimizing an existing application binary for the same instruction set,
- c) replicating a virtual machine so that multiple (possibly different) Oses can be supported simultaneously,
- d) composing virtual machine software to form a more complex, flexible system.

VM software can also provide *replication*, for example by giving a single hardware platform the appearance of multiple platforms (Fig. 4c), each capable of running a complete operating system and/or a set of applications. Finally, the various types of virtual machine software can be composed (Fig. 4d) to form wide variety of architectures, freed of many of the traditional interface constraints.

As just described, 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 that any major innovation, e.g. a new ISA, new OS, or new programming language will be based on VM technology. Consequently, for constructing modern systems, VM “coupling” software has essentially become a fourth major system component, and it merits discussion and study as a discipline -- to the same degree as hardware, operating systems and application software. Because the various virtual machine architectures and underlying technologies have been developed (often independently) by different groups: OS designers, hardware architects, and programming language designers, it is especially important to unify this body of knowledge and understand the base technologies that cut across the various forms of VMs. The goals of this book are to describe the variety of virtual machines in a unified way, to discuss the common underlying technologies that support them, and to explore their many applications.

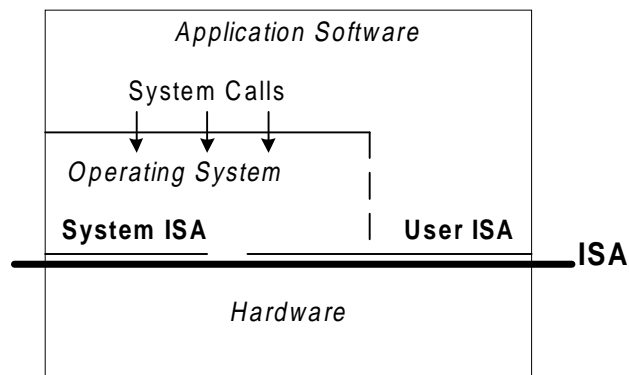
The following subsections briefly describe some of the more important virtual machine architectures and applications.

2 Key System Interfaces

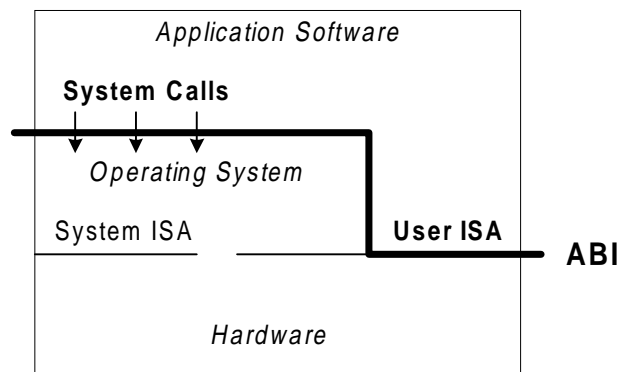
The key interfaces in which we are interested are shown in more detail in Fig. 5. The ISA, shown in Fig. 5a, includes both *user* and *system* instructions. The user instructions are available to both application programs and to the operating system. The system instructions include privileged operations that permit the direct manipulation, allocation, and observation of shared hardware resources (the processor, memory and I/O).

Fig. 5b 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 tables in memory, following specific conventions that are part of the system call interface.



(a)



(b)

Fig. 5 Important System Interfaces

- a) **Instruction Set Architecture (ISA) interface**
- b) **Application Binary Interface (ABI)**

We are now ready to describe and discuss some specific types of virtual machines. These span a fairly broad spectrum of applications, and we will begin with the first types of VMs to be developed.

3 'Classic' Virtual Machines: Replication

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 application with the illusion of having the operating system and hardware to itself. The operating system timeshares the hardware and manages underlying resources to make this possible. In effect, the OS provides replicated ABI virtual machines for each of the concurrently executing applications (Fig. 6).

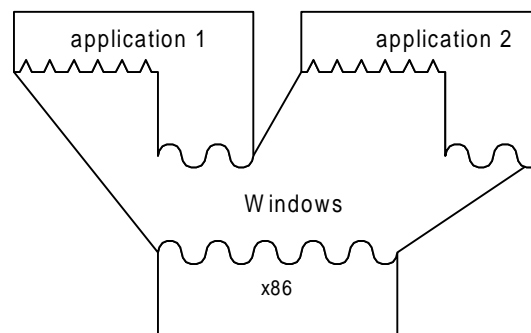


Fig. 6. Conventional multiprogramming provides replicated ABI virtual machines for each of a number of timeshared applications. This replicated VM is so integral to modern OSe that it is not considered to be a separable OS function.

This basic approach was extended from replication at the ABI level to replication at the full ISA level during the 1960s and early 1970s. The resulting VMs could support multiple OS environments simultaneously on the same hardware. At that time, 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 OSe to be run on the shared hardware, and VMs allowed them to do so. Alternatively, a multiplicity of single-user OSe allowed a convenient way of implementing time-sharing amongst several users. Fig. 7 illustrates these classical VMs.

For these systems, replication is the major feature provided by the VM software. The central problem in implementing VMs that replicate the full ISA is dividing a single set of hardware resources among multiple operating system environments. Because of its OS-like functionality, the virtual machine software in these systems is typically referred to as a *Virtual*

Machine Monitor (VMM). 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.

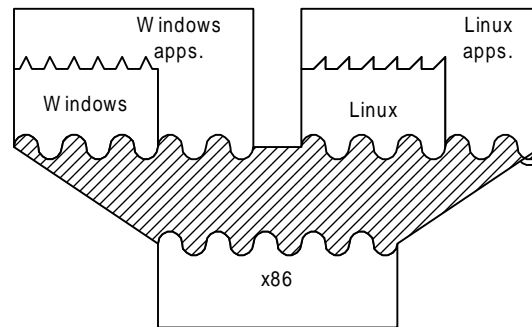


Fig. 7 An example OS VM -- supporting multiple OS environments on the same hardware.

Multiple guest OSes can be supported; in the figure there are two. Each guest OS plus associated applications has the illusion of executing on its own "virtual machine", unaware of the underlying VMM or any other virtual machines with which it shares 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, 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 today's IBM mainframes. Currently, OS VMs are also becoming increasingly important for Intel x86-based PC platforms.

A key issue in implementing a replicated VM is whether the VMM can easily intercept and implement all of the guest OS's actions that interact with hardware resources -- *in a completely transparent way*. That is, whether hardware is easily *virtualizable*. The virtualizability of ISAs was a subject of study in the early 1970s, and some of the evolutionary enhancements to the IBM mainframe architectures have been targeted at improved virtualizability. On the other hand, the Intel X86 instruction set lacks many virtualizability features and requires significant software emulation support.

It is important to note that Fig. 7 only illustrates the functionality of a replicated VM, the actual implementation may have a somewhat different structure. In particular, the VMM in Fig. 7 seems to sit directly on bare hardware and all the OSes must go through it to access hardware. Alternative implementations, however, may place the VMM software on top of an existing “host” OS and rely on it to provide some services on behalf of other “guest” OSes. This hosted VM approach is taken in the VMware implementation, a modern VM that replicates x86 hardware platforms.

4 ABI VMs: Translation and Optimization

In many instances, what is really of interest to users is portability of applications – not OSes; supporting the full OS in the VM is a means to this end. However, there are other approaches. In particular, some VMs work at the ABI level.

An example ABI VM is illustrated in Fig. 8. Application programs are compiled for a *guest ISA*, but the hardware implements a different *native ISA*. As shown in the figure, the operating system executes the native ISA. The example illustrates the Compaq FX!32 system. The FX!32 system runs Intel x86 application binaries compiled for Windows NT, on a Compaq Alpha hardware platform also running Windows NT.

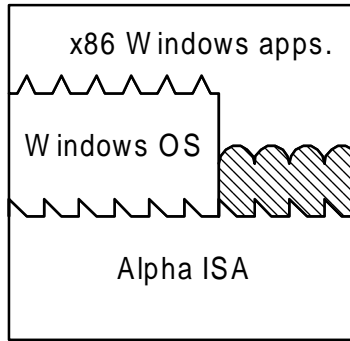


Fig. 8. An ABI VM with emulation/translation of guest applications. The Digital/Compaq FX!32 system allows Windows x86 applications to be run on an Alpha Windows platform.

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

For better performance, *binary translation* is typically used. Groups of guest 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.

Interpretation and binary translation have different performance characteristics. Interpretation has relatively low startup overhead but consumes significant time whenever an instruction is emulated. Conversely, binary translation has high initial overhead when performing the translations, but it is fast for each repeated execution. Consequently, some virtual machines use a phased emulation model combined with profiling. Initially, a block of guest 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.

For handling OS calls in an ABI VM the OS calls made by the guest must be translated to the OS calls of the host. If the OSes are the same, as is the case with FX!32, this is relatively straightforward: essentially all that has to be done is a translation between interface protocols. If

the OSes are different, then it becomes necessary to implement code that can convert guest OS calls into equivalent host OS calls (possibly several host calls plus other emulation code may be needed to implement one guest call.) For example, the Sun Microsystems WABI system that allows Windows ABIs to be executed on a Solaris system takes this latter approach.

5 Whole System VMs: Translation

In the classic replicated VMs described in Section 3, all the operating systems and applications use the same ISA as the underlying hardware. Frequently, however, the host and guest systems have neither in common. For example, the Apple Macintosh and Windows PC use different ISAs and OSes, even though they are the two most popular systems today. As another example, Sun Microsystems servers use a different OS and ISA than the windows PCs that are commonly attached as clients.

This leads us to a type of translating *VMs*, where a complete software system, both OS and applications, are supported on a host system supporting a different ISA and OS. These can also be called "whole system" VMs because they support all software. Because the ISAs are different, both application and OS code require emulation, e.g. via binary translation. Furthermore, the guest operating system interaction with hardware resources must be done through a VMM that either runs directly on the hardware or which is supported by a conventional host OS running on the hardware. We consider the latter case first.

Fig. 9 shows 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, i.e. to mix applications for both ISAs; this feature is illustrated in the right section of the drawing.

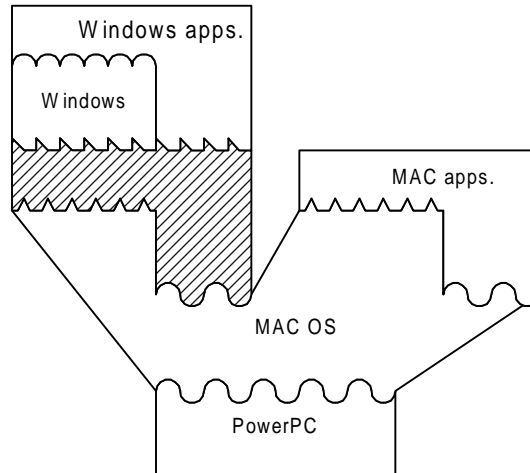


Fig. 9. A whole system VM that supports a guest OS and applications, in addition to host applications.

To implement a 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.

Alternatively, one could implement a VMM in a manner similar to the implementation of the VMM in the traditional OS VMs illustrated earlier in Fig. 7. Here, the VMM has direct control over hardware resources and is free to use the system operations belonging to the system ISA. This allows a more flexible translation of the guest application and OS software. It also allows a more direct path for controlling the host system's hardware resources. Meanwhile, such a VMM can be written not only to support guest environments using different ISA, it can easily (and simultaneously) support other guest OS(es) using the same ISA as the underlying hardware -- just as in the traditional OS VMs.

6 High Level VMs: Complete Platform Independence

For the VMs described above, cross-platform portability is clearly a very important objective. For example, the Compaq FX!32 enabled portability of application software compiled for a popular platform (x86 PC) to a less popular platform (Compaq Alpha).

This objective can be carried further by making application software *independent* of hardware platform, i.e. enable it to run on any and all hardware platforms. One way of accomplishing this is to take a higher-level view and design a VM at the time the application development environment, including a high level language (HLL), is 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 ABI VMs described above. They focus on supporting applications, not operating systems. And they tend to minimize hardware related-features because these would compromise platform independence.

This type of virtual machine first became popular with the Pascal programming environment. In a conventional system, Fig. 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 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 a different platform, however, it must be re-compiled for that platform.

With Pascal, this model was changed (Fig. 10b). The steps are similar to the conventional ones, but the point at which distribution takes place is at a higher level. In Fig. 10b, 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.

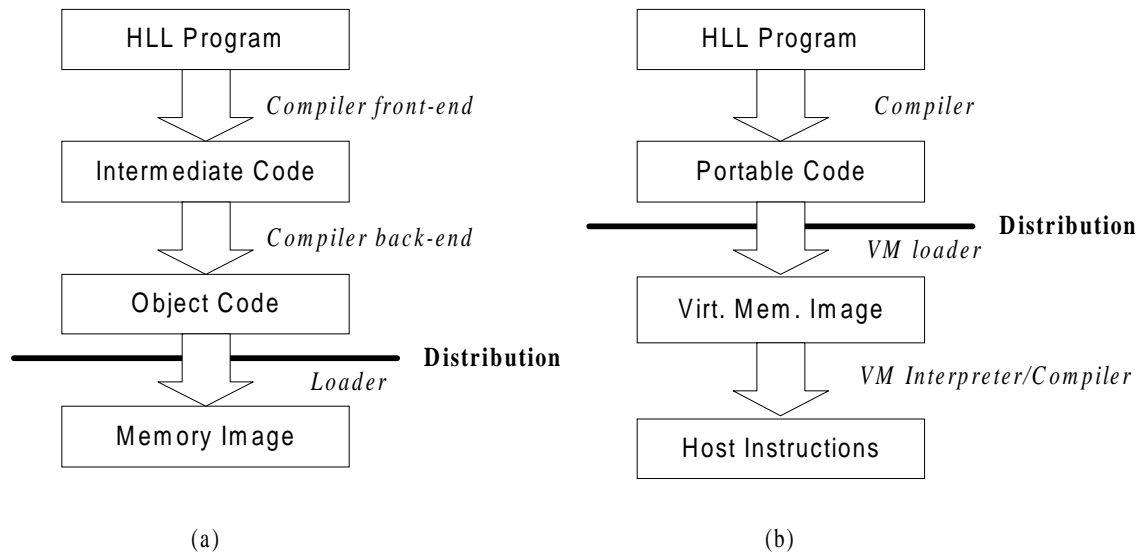


Fig. 10. High Level Language environments

- 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 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.

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 native libraries whose calling interface and functions are part of the Java virtual machine architecture.

7 Co-Designed VMs: Optimization

In all the VM models discussed thus far, the goal has been flexibility and portability -- to either support multiple (different) OSes on the same hardware, or to support different ISAs and OSes on the same hardware. In practice, these virtual machines have been 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 typically the major 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, and/or fault tolerance. VMs of this type may be based on hardware that uses a non-standard, proprietary ISA. In fact, the native ISA is co-developed with the hardware implementation to enhance and enable better hardware implementation features. This native ISA may be completely hidden from all conventional software that runs on the processor.

In general, the native 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 (Fig. 11), 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 "hidden" from all application and system software.

This hidden 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 hidden 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 hidden 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 may be coupled with code optimization. Optimization can be performed at the translation time, and/or it can be performed as an on-going process as a program runs, depending on which code sequences are more frequently executed.

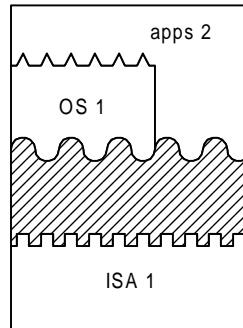


Fig. 11. Co-Designed VM -- VM software translates and executes code for a hidden native ISA.

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.

Note that the ISA translation/optimization functions and resource management functions are in some sense orthogonal and can be used independently. And neither requires the wholesale development of a new proprietary ISA. For example, one can provide hardware monitoring and adaptive resource management functions by extending an existing instruction set without any translation of existing application binaries.

8 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: ISA VMs and ABI VMs. In the first type, the VM supports a complete ISA – both user and system instructions; in the second the VM supports the ABI – user instructions plus system/library calls. Other parts of the taxonomy are based on two differentiating features:

- 1) Whether the guest and host use the same ISA,
- 2) Whether the guest and host use the same OS system calls.

With this taxonomy, Fig. 12 shows the "space" of virtual machines that we have identified.

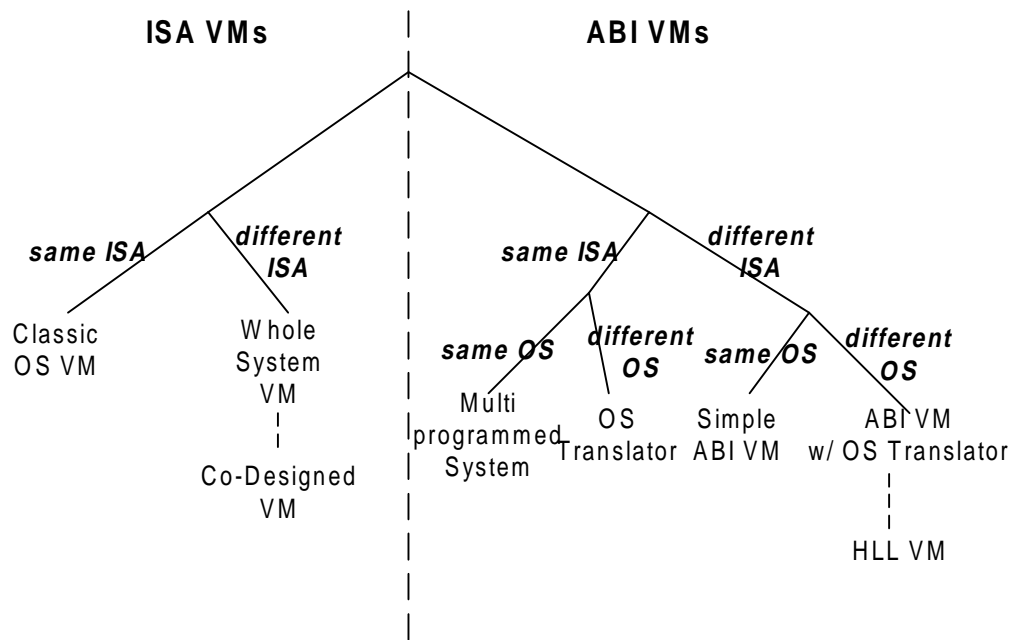


Fig. 12. A taxonomy of virtual machine architectures.

The left-most VM is the classic VM that supports multiple OSES (and their applications) simultaneously. All the VMs execute the same ISA as the underlying hardware, however, so the VM software *replicates* the underlying hardware by virtualizing the hardware resources. These VMs provide a diversity of operating systems to the user(s) and also provide them with a “firewall” level of security from each other.

To support a whole system VM, the underlying VM software uses *translation* to support a completely different ISA. Here, all aspects of the all aspects of the system must be emulated, and most of the VM implementation challenges involve providing good performance while emulation is being performed. Because the entire ISA is supported, any OS that has been implemented for that ISA will run.

Proceeding to the right half of the taxonomy, ABI VMs only have to provide application programs with a functionally correct environment. This means that system calls must produce the results expected by the application code, but the guest operating system software does not have to be directly supported by the VM software. The simplest (and most common) VM is where the ABI to be supported uses both the same ISA and OS as the underlying host system. In fact, this is what is done in a conventional multiprogrammed system – each independent program “thinks” it is operating within its own virtual machine environment. In addition to *replication* provided by multiprogramming, additional VM software may be added to this type of system to provide *optimization*, as is done in the HP Dynamo system.

If the ABI is constructed for a different OS than the underlying host, then some kind of translation of system calls must take place. That is, the VM software must intercept guest OS calls and convert them into equivalent host call(s). If the ISA of both the host and guest are the same, then this is all the VM software must do, and in the taxonomy, we refer to this simply as “OS Translation”. The most challenging part of these VMs is to provide a perfectly correct translation of OS calls.

If the ISA is different in the guest and host, then some form of emulation must take place. Consequently, for implementing these VMs, a key issue is instruction set emulation via interpretation, translation, and optimization. In these VMs, the OSes may be the same for both the host and guest, in which case the translation of OS calls is a straightforward process.

The rightmost section of the taxonomy diagram corresponds to ISA VMs where both the OS and the ISA differ between the guest and host. In many respects these are the most challenging VMs to implement, for both performance reasons (due to ISA translation) and for correctness reasons (they also need to have code that translates at the system call interface).

Co-designed VMs and HLL VMs also fit into the taxonomy in an interesting way. The co-designed VM places a software layer *below* the conventional ISA interface. This software layer is for the benefit of hardware implementers; no conventional software has to be changed. The native

hardware ISA is specially developed for features of a specific hardware implementation. On the other hand, a HLL VM places an additional software layer *above* the conventional ISA interface. This software layer is for the benefit of software implementers, typically to enhance portability. Co-designed VMs are a form of whole system VM because they must support the entire ISA. HLL VMs, on the other hand are targeted only at supporting ABIs – the key interface for application software.

A good way to end this summary is with an example of a realistic system that could conceivably be in use today (Fig. 13). The example clearly illustrates power of composing VMs. Someone 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 in fact executing as native VLIW.

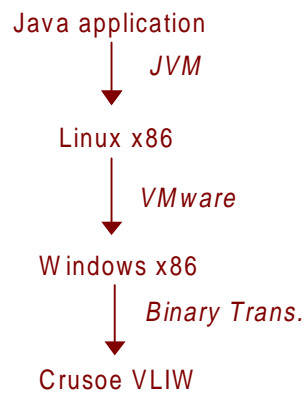


Fig. 13. Three levels of VMs: a Java application running on a Java VM, running on an OS VM, running on a co-designed VM.