COVER FEATURE

The Architecture of Virtual Machines

A virtual machine can support individual processes or a complete system depending on the abstraction level where virtualization occurs. Some VMs support flexible hardware usage and software isolation, while others translate from one instruction set to another.

James E. Smith
University of Wisconsin-Madison

Ravi Nair
IBM T.J. Watson Research Center

Virtualization has become an important tool in computer system design, and virtual machines are used in a number of subdisciplines ranging from operating systems to programming languages to processor architectures. By freeing developers and users from traditional interface and resource constraints, VMs enhance software interoperability, system impregnability, and platform versatility.

Because VMs are the product of diverse groups with different goals, however, there has been relatively little unification of VM concepts. Consequently, it is useful to take a step back, consider the variety of VM architectures, and describe them in a unified way, putting both the notion of virtualization and the types of VMs in perspective.

ABSTRACTION AND VIRTUALIZATION

Despite their incredible complexity, computer systems exist and continue to evolve because they are designed as hierarchies with *well-defined interfaces* that separate *levels of abstraction*. Using well-defined interfaces facilitates independent subsystem development by both hardware and software design teams. The simplifying abstractions hide lower-level implementation details, thereby reducing the complexity of the design process.

Figure 1a shows an example of abstraction applied to disk storage. The operating system abstracts hard-disk addressing details—for example, that it is comprised of sectors and tracks—so that the disk appears to application software as a set of variable-sized files. Application programmers can then create, write, and read files without knowing the hard disk's construction and physical organization.

A computer's instruction set architecture (ISA) clearly exemplifies the advantages of well-defined interfaces. Well-defined interfaces permit development of interacting computer subsystems not only in different organizations but also at different times, sometimes years apart. For example, Intel and AMD designers develop microprocessors that implement the Intel IA-32 (x86) instruction set, while Microsoft developers write software that is compiled to the same instruction set. Because both groups satisfy the ISA specification, the software can be expected to execute correctly on any PC built with an IA-32 microprocessor.

Unfortunately, well-defined interfaces also have their limitations. Subsystems and components designed to specifications for one interface will not work with those designed for another. For example, application programs, when distributed as compiled binaries, are tied to a specific ISA and depend on a specific operating system interface. This lack of interoperability can be confining, especially in a world of networked computers where it is advantageous to move software as freely as data.

Virtualization provides a way of getting around such constraints. Virtualizing a system or component—such as a processor, memory, or an I/O device—at a given abstraction level maps its interface and visible resources onto the interface and resources of an underlying, possibly different, real system. Consequently, the real system appears as a different virtual system or even as multiple virtual systems.

Unlike abstraction, virtualization does not necessarily aim to simplify or hide details. For example, in Figure 1b, virtualization transforms a single large disk into two smaller virtual disks, each of which

Published by the IEEE Computer Society
0018-9162/05/$20.00 © 2005 IEEE

SAMSUNG 1067

appears to have its own tracks and sectors. Virtualizing software uses the file abstraction as an intermediate step to provide a mapping between the virtual and real disks. A write to a virtual disk is converted to a file write (and therefore to a real disk write). Note that the level of detail provided at the virtual disk interface—the sector/track addressing—is no different from that for a real disk; no abstraction takes place.

VIRTUAL MACHINES

The concept of virtualization can be applied not only to subsystems such as disks but to an entire machine. To implement a virtual machine, developers add a software layer to a real machine to support the desired architecture. By doing so, a VM can circumvent real machine compatibility and hardware resource constraints.

Architected interfaces

A discussion of VMs is also a discussion about computer architecture in the pure sense of the term. Because VM implementations lie at architected interfaces, a major consideration in the construction of a VM is the fidelity with which it implements these interfaces.

Architecture, as applied to computer systems, refers to a formal specification of an interface in the system, including the logical behavior of resources managed via the interface. *Implementation* describes the actual embodiment of an architecture. Abstraction levels correspond to implementation layers, whether in hardware or software, each associated with its own interface or architecture.

Figure 2 shows some important interfaces and implementation layers in a typical computer system. Three of these interfaces at or near the HW/SW boundary—the instruction set architecture, the application binary interface, and the application programming interface—are especially important for VM construction.

Instruction set architecture. The ISA marks the division between hardware and software, and consists of interfaces 3 and 4 in Figure 2. Interface 4 represents the user ISA and includes those aspects visible to an application program. Interface 3, the system ISA, is a superset of the user ISA and includes those aspects visible only to operating system software responsible for managing hardware resources.

Application binary interface. The ABI gives a program access to the hardware resources and services available in a system through the user ISA (interface 4) and the system call interface (interface 2).

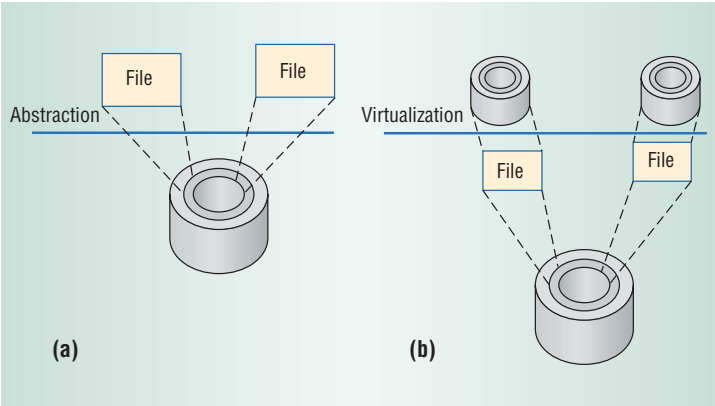

Figure 1. Abstraction and virtualization applied to disk storage. (a) Abstraction provides a simplified interface to underlying resources. (b) Virtualization provides a different interface or different resources at the same abstraction level.

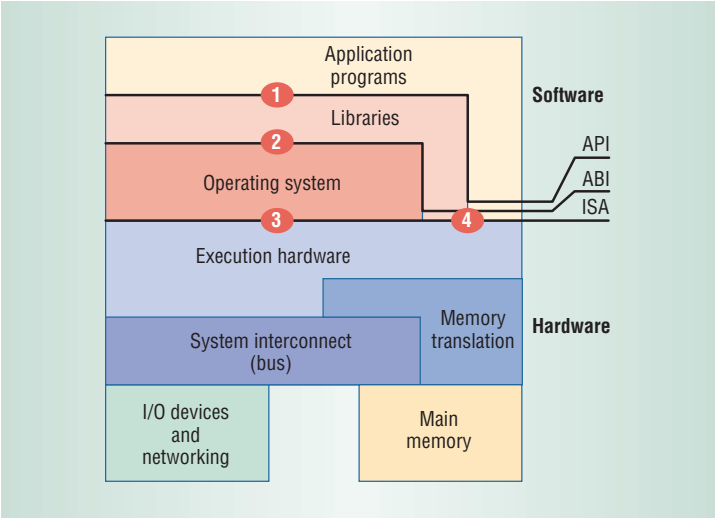

Figure 2. Computer system architecture. Key implementation layers communicate vertically via the instruction set architecture (ISA), application binary interface (ABI), and application programming interface (API).

The ABI does not include system instructions; rather, all application programs interact with the hardware resources indirectly by invoking the operating system's services via the system call interface. System calls provide a way for an operating system to perform operations on behalf of a user program after validating their authenticity and safety.

Application programming interface. The API gives a program access to the hardware resources and services available in a system through the user ISA (interface 4) supplemented with high-level language

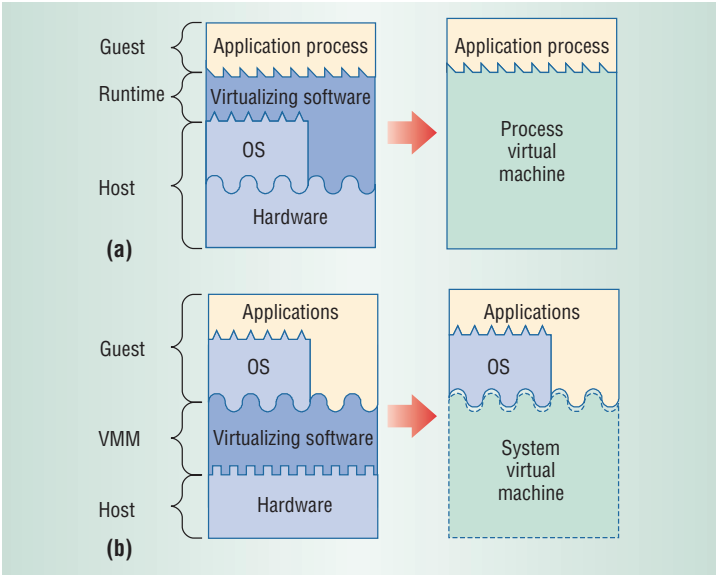

Figure 3. Process and system VMs. (a) In a process VM, virtualizing software translates a set of OS and user-level instructions composing one platform to those of another. (b) In a system VM, virtualizing software translates the ISA used by one hardware platform to that of another.

(HLL) library calls (interface 1). Any system calls are usually performed through libraries. Using an API enables application software to be ported easily, through recompilation, to other systems that support the same API.

Process and system VMs

To understand what a virtual machine is, it is first necessary to consider the meaning of "machine" from both a process and system perspective.

From the perspective of a process executing a user program, the machine consists of a logical memory address space assigned to the process along with user-level instructions and registers that allow the execution of code belonging to the process. The machine's I/O is visible only through the operating system, and the only way the process can interact with the I/O system is through operating system calls. Thus the ABI defines the machine as seen by a process. Similarly, the API specifies the machine characteristics as seen by an application's HLL program.

From the perspective of the operating system and the applications it supports, the entire system runs on an underlying machine. A system is a full execution environment that can support numerous processes simultaneously. These processes share a file system and other I/O resources. The system environment persists over time as processes come and go. The system allocates real memory and I/O resources to the processes, and allows the processes to interact with their resources. From the system perspective, therefore, the underlying hardware's characteristics alone define the machine; it is the ISA that provides the interface between the system and machine.

Just as there are process and system perspectives of "machine," there are process and system virtual machines. A *process* VM is a virtual platform that executes an individual process. This type of VM exists solely to support the process; it is created when the process is created and terminates when the process terminates. In contrast, a *system* VM provides a complete, persistent system environment that supports an operating system along with its many user processes. It provides the guest operating system with access to virtual hardware resources, including networking, I/O, and perhaps a graphical user interface along with a processor and memory.

The process or system that runs on a VM is the *guest,* while the underlying platform that supports the VM is the *host*. The virtualizing software that implements a process VM is often termed the *runtime,* short for "runtime software." The virtualizing software in a system VM is typically referred to as the *virtual machine monitor* (VMM).

Figure 3 depicts process and system VMs, with compatible interfaces illustrated graphically as meshing boundaries. In a process VM, the virtualizing software is at the ABI or API level, atop the OS/HW combination. The runtime emulates both user-level instructions and either operating system or library calls. In a system VM, the virtualizing software is between the host hardware machine and the guest software. The VMM emulates the hardware ISA so that the guest software can potentially execute a different ISA from the one implemented on the host. However, in many system VM applications, the VMM does not perform instruction emulation; rather, its primary role is to provide virtualized hardware resources.

PROCESS VIRTUAL MACHINES

Process VMs provide a virtual ABI or API environment for user applications. In their various implementations, process VMs offer replication, emulation, and optimization.

Multiprogrammed systems

The most common process VM is so ubiquitous that few regard it as being a VM. Most operating systems can simultaneously support multiple user processes through multiprogramming, which gives each process the illusion of having a complete machine to itself. Each process has its own address space, registers, and file structure. The operating system time-shares the hardware and manages underlying resources to make this possible. In effect, the operating system provides a replicated

process-level VM for each of the concurrently executing applications.

Emulators and dynamic binary translators

A more challenging problem for process-level VMs is that of supporting program binaries compiled to an instruction set different from the one the host executes. A recent example of a process VM is the Intel IA32-EL,[1] which allows Intel IA-32 application binaries to run on Itanium hardware.

The most straightforward way of performing emulation is through *interpretation*. An interpreter program fetches, decodes, and emulates the execution of individual guest instructions. This can be a relatively slow process, requiring tens of host instructions for each source instruction interpreted. Better performance can be obtained through *dynamic binary translation,* which converts guest instructions to host instructions in blocks rather than instruction by instruction and saves them for reuse in a software cache. Repeated execution of the translated instructions thus amortizes the relatively high overhead of translation.

Same-ISA binary optimizers

To reduce performance losses, dynamic binary translators sometimes perform code optimizations during translation. This capability leads naturally to VMs wherein the instruction sets that the host and guest use are the same, with optimization of a program binary as the VM's sole purpose. Same-ISA dynamic binary optimizers use profile information collected during the interpretation or translation phase to optimize the binary on-the-fly. An example of such an optimizer is the Dynamo system, originally developed as a research project at Hewlett-Packard.[2]

High-level-language VMs

For process VMs, cross-platform portability is clearly a key objective. However, emulating one conventional architecture on another provides cross-platform compatibility only on a case-by-case basis and requires considerable programming effort. Full cross-platform portability is more readily achieved by designing a process-level VM as part of an overall HLL application development environment. The resulting HLL VM does not directly correspond to any real platform; rather, it is designed for ease of portability and to match the features of a given HLL or set of HLLs.

Figure 4 shows the difference between a conventional platform-specific compilation environment and an HLL VM environment. In a conventional system, shown in Figure 4a, a compiler front end first generates intermediate code that is similar to machine code but more abstract. Then, a code generator uses the intermediate code to generate a binary containing machine code for a specific ISA and operating system. This binary file is distributed and executed on platforms that support the given ISA/OS combination.

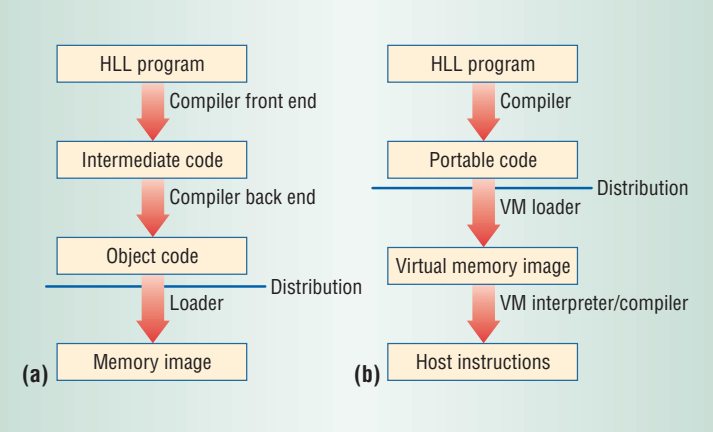

Figure 4. High-level-language environments. (a) Conventional environment where platform-dependent object code is distributed. (b) HLL VM environment where a platform-dependent VM executes portable intermediate code.

In an HLL VM, as shown in Figure 4b, a compiler front end generates abstract machine code in a virtual ISA that specifies the VM's interface. This virtual ISA code, along with associated data structure information (metadata), is distributed for execution on different platforms. Each host platform implements a VM capable of loading and executing the virtual ISA and a set of library routines specified by a standardized API. In its simplest form, the VM contains an interpreter. More sophisticated, higher-performance VMs compile the abstract machine code into host machine code for direct execution on the host platform.

An advantage of an HLL VM is that application software is easily ported once the VM and libraries are implemented on a host platform. While the VM implementation takes some effort, it is much simpler than developing a full-blown compiler for a platform and porting every application through recompilation.

The Sun Microsystems Java VM architecture[3] and the Microsoft Common Language Infrastructure, which is the foundation of the .NET framework,[4] are widely used examples of HLL VMs. The ISAs in both systems are stack-based to eliminate register requirements and use an abstract data specification and memory model that supports secure object-oriented programming.

SYSTEM VIRTUAL MACHINES

A system VM provides a complete environment in which an operating system and many processes,

VM technology provides isolation between multiple systems running concurrently on the same hardware platform.

possibly belonging to multiple users, can coexist. By using system VMs, a single-host hardware platform can support multiple, isolated guest operating system environments simultaneously.

System VMs emerged during the 1960s and early 1970s[5] and were the origin of the term virtual machine. At that time, mainframe computer systems were very large, expensive, and usually shared among numerous users; with VM technology, different user groups could run different operating systems on the shared hardware.

As hardware became less expensive and much of it migrated to the desktop, interest in these original system VMs faded. Today, however, system VMs are enjoying renewed popularity as the large mainframe systems of the past have been replaced by servers or server farms shared by many users or groups.

Perhaps the most important current application of system VM technology is the isolation it provides between multiple systems running concurrently on the same hardware platform. If security on one guest system is compromised or if one guest operating system suffers a failure, the software running on other guest systems is not affected.

In a system VM, the VMM primarily provides platform replication. The central issue is dividing a set of hardware resources among multiple guest operating system environments—an example is disk virtualization, as in Figure 1. The VMM has access to, and manages, all the hardware resources. A guest operating system and its application processes are then managed under (hidden) control of the VMM. When a guest operating system performs a privileged instruction or operation that directly interacts with shared hardware resources, the VMM intercepts the operation, checks it for correctness, and performs it on behalf of the guest. Guest software is unaware of this behind-the-scenes work.

Classic system VMs

From the user perspective, most system VMs provide essentially the same functionality but differ in their implementation details. The classic approach[6] places the VMM on bare hardware and the VMs fit on top. The VMM runs in the most highly privileged mode, while all guest systems run with reduced privileges so that the VMM can intercept and emulate all guest operating system actions that would normally access or manipulate critical hardware resources.

Hosted VMs

An alternative system VM implementation builds virtualizing software on top of an existing host operating system, resulting in a *hosted* VM. An advantage of a hosted VM is that a user installs it just like a typical application program. Further, virtualizing software can rely on the host operating system to provide device drivers and other lower-level services rather than on the VMM. An example of a hosted VM implementation is the VMware GSX server,[7] which runs on IA-32 hardware platforms.

Whole-system VMs

In conventional system VMs, all guest and host system software as well as application software use the same ISA as the underlying hardware. In some situations, however, the host and guest systems do not have a common ISA. For example, the two most popular desktop systems today, Windows PCs and Apple PowerPC-based systems, use different ISAs (and different operating systems).

Whole-system VMs deal with this situation by virtualizing all software, including the operating system and applications. Because the ISAs differ, the VM must emulate both the application and operating system code. An example of this type of VM is the Virtual PC,[8] in which a Windows system runs on a Macintosh platform. The VM software executes as an application program supported by the host operating system and uses no system ISA operations.

Multiprocessor virtualization

An interesting form of system virtualization occurs when the underlying host platform is a large shared-memory multiprocessor. Here, an important objective is to partition the large system into multiple smaller multiprocessor systems by distributing the underlying hardware resources of the large system.

With *physical partitioning,*[9] the physical resources that one virtual system uses are disjoint from those used by other virtual systems. Physical partitioning provides a high degree of isolation, so that neither software problems nor hardware faults on one partition affect programs in other partitions. With *logical partitioning,*[10] the underlying hardware resources are time-multiplexed between the different partitions, thereby improving system resource utilization. However, some of the benefits of hardware isolation are lost.

Both partitioning techniques typically use special software or firmware support based on underlying hardware modifications specifically targeted at partitioning.

Codesigned VMs

Functionality and portability are the goals of most system VMs that are implemented on hardware already developed for some standard ISA. In contrast, *codesigned* VMs implement new, proprietary ISAs targeted at improving performance, power efficiency, or both. The host's ISA may be completely new, or it may be an extension of an existing ISA.

A codesigned VM has no native ISA applications. Instead, the VMM appears to be part of the hardware implementation; its sole purpose is to emulate the guest's ISA. To maintain this illusion, the VMM resides in a region of memory concealed from all conventional software. It includes a binary translator that converts guest instructions into optimized sequences of host ISA instructions and caches them in the concealed memory region.

Perhaps the best-known example of a codesigned VM is the Transmeta Crusoe.[11] In this processor, the underlying hardware uses a very-long instruction word architecture, and the guest ISA is the Intel IA-32. The Transmeta designers focused on the power-saving advantages of simpler VLIW hardware.

The IBM AS/400 (now the iSeries) also uses codesigned VM techniques.[12] Unlike other codesigned VMs, the AS/400's primary design objective is to provide support for an object-based instruction set that redefines the HW/SW interface in a novel fashion. Current AS/400 implementations are based on an extended PowerPC ISA, whereas earlier versions used a considerably different, proprietary ISA.

VIRTUAL MACHINE TAXONOMY

Given this broad array of VMs, with different goals and implementations, it is helpful to put them in perspective and organize the common implementation issues. Figure 5 presents a simple taxonomy of VMs, which are first divided into either process or system VMs. Within these two major categories, VMs can be further distinguished according to whether they use the same ISA or a different one. The basis for this differentiation is that ISA emulation is a dominant feature in those VMs that support it.

Among the process VMs that do not perform ISA emulation are multiprogrammed systems, which most of today's computers already support. Also included are same-ISA dynamic binary optimizers, which employ many of the same techniques as ISA emulation.

Process VMs with different guest and host ISAs include dynamic translators, with the machine interface typically defined at the ABI level, and HLL VMs with an API-level interface.

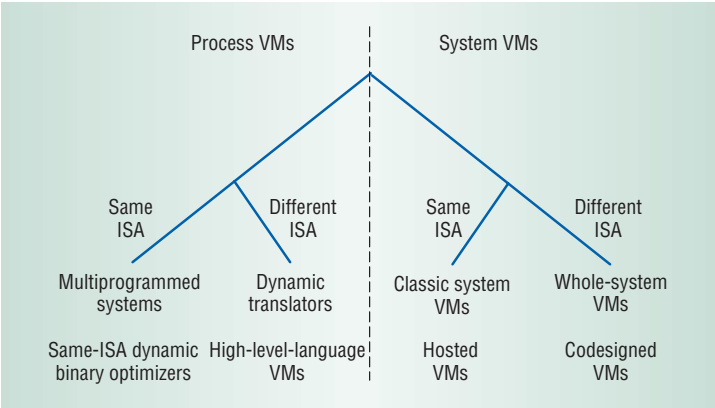

Figure 5. Virtual machine taxonomy. Within the general categories of process and system VMs, ISA simulation is the major basis of differentiation.

System VMs consist of classic system VMs as well as hosted VMs that provide replicated, isolated system environments. The primary difference between classic system and hosted VMs is the VMM implementation rather than the function they provide to the user.

In whole-system VMs, wherein the guest and host ISAs are different, performance is often secondary to accurate functionality. When performance or power/area efficiency becomes important, as is the case with codesigned VMs, the VM implementation interface may be closer to the processor's physical hardware.

Modern computer systems are complex structures containing numerous closely interacting components in both software and hardware. Within this universe, virtualization acts as a type of interconnection technology. Interjecting virtualizing software between abstraction layers near the HW/SW interface forms a virtual machine that allows otherwise incompatible subsystems to work together. Further, replication by virtualization enables more flexible and efficient use of hardware resources.

VMs are now widely used to enable interoperability between hardware, system software, and application software. Given the heavy reliance on standards and consolidation occurring in the industry, it is likely that any new ISA, operating system, or programming language will be based on VM technology. In the future, VMs should be viewed as a unified discipline to the same degree that hard-

ware, operating systems, and application software are today. ■

References

1. L. Baraz et al., "IA-32 Execution Layer: A Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2003, pp. 191-204.
2. V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation,* ACM Press, 2000, pp. 1-12.
3. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* 2nd ed., Addison-Wesley, 1999.
4. D. Box, *Essential .NET, Volume 1: The Common Language Runtime,* Addison-Wesley, 2002.
5. R.J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Research and Development,* Sept. 1981, pp. 483-490.
6. R.P. Goldberg, "Survey of Virtual Machine Research," *Computer,* June 1974, pp. 34-35.
7. J. Sugerman, G. Venkitachalam, and B-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," *Proc. General Track: 2001 Usenix Ann. Technical Conf.,* Usenix Assoc., 2001, pp. 1-14.
8. E. Traut, "Building the Virtual PC," *Byte,* Nov. 1997, pp. 51-52.
9. Sun Microsystems, "Sun Enterprise 10000 Server: Dynamic System Domains," tech. white paper, 1999; www.sun.com/datacenter/docs/domainswp.pdf.
10. T.L. Borden, J.P. Hennessy, and J.W. Rymarczyk, "Multiple Operating Systems on One Processor Complex," *IBM Systems J.,* Jan. 1989, pp. 104-123.
11. A. Klaiber, "The Technology Behind Crusoe Processors: Low-Power x86-Compatible Processors Implemented with Code Morphing Software," tech. brief, Transmeta Corp., 2000.
12. F.G. Soltis, *Inside the AS/400,* Duke Press, 1996.

James E. Smith *is a professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His current research interests include high-performance and power-efficient processor implementations, processor performance modeling, and virtual machines. Smith received a PhD in computer science from the University of Illinois. He is a member of the IEEE and the ACM. Contact him at jes@ece.wisc.edu.*

Ravi Nair *is a research staff member at the IBM T.J. Watson Research Center. His current interests include processor microarchitectures, dynamic compilation, and virtual machine technology. Nair received a PhD in computer science from the University of Illinois. He is a Fellow of the IEEE and a member of the IBM Academy of Technology. Contact him at nair@us.ibm.com.*