ON THE RELATIONSHIP BETWEEN VIRTUAL MACHINES AND EMULATORS

Efrem G. Mallach
Honeywell Information Systems
Billerica, Massachusetts

ABSTRACT

The subjects of virtual machines and emulators have been treated as entirely separate. The purpose of this paper is to show that they have much in common. Not only do the usual implementations have many shared characteristics, but this commonality extends to the theoretical concepts on which they are based; the concepts of memory mapping and I/O operation simulation are discussed to emphasize this. The paper then discusses structural issues, and points out why the question of instruction set is becoming less valid as a point of distinction between the concepts. Possible combinations of virtual machines and emulators are discussed. In conclusion, it is recommended that workers in both fields keep the relationship between the two in mind.

Historically, the subjects of virtual machines and emulators have been treated as entirely unrelated. Authors have addressed one or the other, but not both. Work has taken place with little or no interaction. Emulator developers have not taken advantage of theoretical advances in the field of virtual machines, and virtual machine developers have not utilized the wealth of practical experience that exists for emulators.

This paper is intended to show that these two concepts share much common ground. While the initial objectives and ultimate uses of the two types of systems may be quite different, many concepts are shared. Regarding both virtual machines and emulators as manifestations of a single notion can provide valuable insight. Workers in either area can benefit from greater awareness of developments in the other. Our purpose in pointing this out is to initiate a dialogue from which both groups can profit.

We begin with typical definitions of the two terms:

1. Emulation, from Lichstein [1]: "...a process whereby one computer is set up to permit the execution of programs written for another computer. This is done with...hardware features...and software..."

2. Virtual machines, from Goldberg [2]: "A system...which...is a hardware-software duplicate of a real existing machine, in which a non-trivial subset of the virtual machine's instructions execute directly on the host machine..."

These definitions appear to have little in common. Definitions of emulation normally focus on the difference between the machine for which some programs were written and the one on which they are to be executed. Definitions of virtual machines, on the other hand, concentrate on the concept of a level of software control between the program and the hardware, and on the mapping of computational entities seen by a program into corresponding entities actually used to support a computation.

Progressing onward from these very different definitions, we consider how systems of these types have typically been implemented; similarities will soon become apparent. The typical virtual machine, be it "family-virtualizing" or "self-virtualizing" in Goldberg's terminology [2], uses hardware-firmware instruction execution facilities for the bulk of its activity. When a "sensitive instruction" (defined by Goldberg [3]) is encountered, supervisory software* is invoked. This software maps the request for input/output services, for changes in the system state, or whatever, into a corresponding (but not necessarily precisely parallel) function, in a manner that is invisible to the program being executed. This approach is unable to execute programs having time dependencies, programs requiring the physical capabilities of devices not present, and so on, but is useful for many programs in practice.

We now consider a typical emulator**. It uses hardware-firmware instruction execution facilities for the bulk of its activity. When a "sensitive instruction" is encountered, emulator software* is invoked. This software maps the request for input/output services, for changes in the system state, or whatever, into a corresponding (but not necessarily precisely parallel) function, in a manner that is invisible to the program being executed. This approach is unable to execute programs having time dependencies, programs requiring the physical capabilities of devices not present, and so on, but is useful for many programs in practice.

The parallelism between the preceding two paragraphs is far from accidental. It is a fact that the implementation of emulators and the implementation of virtual machines have many meaningful similarities. The

* Though software has usually been used here, there is no reason in principle why it must be; a system could support some or all of these mappings in hardware or firmware.

**Emulators can be classified as firmware-controlled, software-controlled, or auxiliary processor. Auxiliary processors are seldom used today, though Honeywell and NCR have been successful with them. Software-controlled emulators, used largely at the upper end of IBM's System/360 and System/370, resemble software simulators helped by special extensions to the host system's instruction set. In this paper we deal specifically with the more common firmware-controlled emulators, though much of the discussion is applicable to the other types. Further clarification and examples can be found in Mallach [4]. Also, we deal here only with integrated emulators, which have largely superseded the stand-alone variety; the distinction is discussed more fully by Allred [5].

concept of trapping input/output instructions, and other instructions that can affect or inspect the state of the system, is common to both. More importantly, the theoretical basis for these concepts is the same for both, as are the specific mapping methods used to support them. To verify this assertion, let us consider two mappings: memory mapping and input/output operation mapping.

Memory mapping is associated with virtual memory. The central concept of virtual memory is the distinction between the address supplied by a program, known as a "logical address", and the memory location which actually supports the reference, known as a "physical address". It is this distinction which permits emulated or virtual machines to operate as if they have full access to physical memory, while in fact they share it with other programs. It is essential that we distinguish between this essence of virtual memory and the many specific implementations of the concept for various purposes. In particular, we must recognize that paging, mapping a continuous logical address space into a discontinuous physical address space, is not essential for virtual memory to exist; neither is segmentation, which supports a discontinuous logical address space. Furthermore, the use of a backing store to hold unneeded portions of a program is not essential to the concept, and it is certainly not essential that the logical address space of a program be much larger than the physical memory of the computer on which it runs. All these are specific implementations of virtual memory, useful in practice, but not conceptually necessary in distilling the essence of virtual memory.

The second and last theoretical requirement that both emulators and virtual machines impose on a virtual memory scheme is that the logical address space supported by the implementation be indistinguishable from the physical address space of the emulated or virtualized machine. This requirement can be met in a number of ways. The fact that virtual machines typically do use backing store management systems and emulators typically don't is not a fundamental distinction. Both emulators and virtual machines require virtual memory, and both impose similar requirements on it.

As for input/output operations, neither current virtual machines nor current emulators can send input/output commands as issued by the program directly to the hardware. (The ability to do so in principle is the same for both.) Virtual machines and emulators are both faced with programs that behave as if they had full control over the hardware. They both must intercept a command with no knowledge of the larger logical context in which it was issued, must determine what real device corresponds to the device called for, and must construct commands that will accomplish the desired objective on that real device. They must both, in many cases, map operations onto devices having meaningful physical differences from the virtual device: for example, a disk file might be used instead of a tape unit, a remote terminal instead of a line printer, and so on. In one special case, virtual machines have an advantage: where the virtual device is similar to the real device, command translation may be trivial. In general, however, the problem faced by the two classes of system is the same: to provide a virtual I/O environment indistinguishable from the real one which the emulated or virtualized programs were designed to deal with.

A distinction is often made between virtual machines and emulators on the basis of the instruction set they support. It might be said, for example, that the instruction set of a virtual machine must be substantially the same as that of the host machine, while that of an emulated machine must be substantially different. (Goldberg, in the definition of "virtual machine" cited incompletely earlier, continues in this vein.) We assert that this is no longer a meaningful distinction. First, the execution of user mode instructions is widely recognized to be the easiest part of building either a virtual machine or an emulator. Also, with microprogramming, it is possible for one machine to support many instruction sets. If appropriate consideration is given to all of them in the hardware, it is possible to support very different instruction sets with comparable efficiency, which makes the question of "native mode" somewhat moot. Another consideration is that machines such as Nanodata's QM-1 [6] provide a means for effectively reconfiguring the microprogrammed base to support new data types, instruction formats, and the like at any time, and thus truly have no "native" instruction set. Finally, the existence of commercial systems such as the Burroughs 1700, which has many "native" instruction sets, makes it impossible to rely on this concept as a rigorous dividing line.

Having considered the similarities between virtual machines and emulators, let us consider the total software structure of a system. The structure of a typical virtual machine system such as CP/67 is shown in Figure 1, ignoring virtual machine recursion for the moment. A typical operating system running a number of integrated emulators is shown in Figure 2.

The major difference between the two figures is that the emulators operate under emulator software packages, while the virtual machines operate directly under the virtual machine monitor. It is sometimes felt that this difference somehow makes virtual machines a "cleaner" concept, while emulators are a bit "messy" by comparison. The importance of this difference is often greatly exaggerated, since at a deeper level the functions performed by the total software in both figures are quite comparable. The difference between the two structures is one of developer choice, and comes about because of the typical development environments and anticipated usage environments to the two types of system. However, the division of functions in Figure 2 into operating system functions and emulator software functions is not inherent in the problem to be solved, and an emulator could easily be structured as shown in Fig. 1 to run many emulated systems under one software package. Conversely, it is quite practical to split virtual machine monitor functions along the lines of Fig. 2, and this has been done in at least three instances [7, 8, 9]. We thus see that similar architectures are quite feasible for virtual machines and emulators. Furthermore, increasing acceptance of shared code and of distributed operating system functions may blur the structural differences between the two figures still more.

We now remedy our earlier omission, and consider multiple levels of virtual machines and emulators. There are four possible cases:

1. Virtual machine under virtual machine. This has been done by IBM.

2. Emulator under virtual machine. This has been done by IBM.

3. Virtual machine under emulator. It is clearly possible in principle to emulate, say, a 360/67. Having done so, it would be possible to run CP/67 on the emulator.

4. Emulator under emulator. This has never been done because it is usually more practical to implement two separate emulators. However, there is no theoretical impediment.

The point we wish to emphasize here is that it is conceptually quite feasible to have a multi-level system, in which some levels are considered virtual machines and some are considered emulators in the traditional sense. Once this is done, the distinction between virtual machines and emulators will become quite blurred. We will have a computing system in which various program exeuction environments, supporting different machine languages, are in turn supported by lower-level software structures implemented in different machine languages. The matter of whether the lower-level structure happens to use the same instruction set as the execution environment it supports will cease to be an issue. Such a situation may well become common in fourth-generation systems.

We may conclude that the distinction between virtual machines and emulators is largely a semantic and a historical one. An emulator might well be defined as "a virtual machine, the instruction set of which is not the one in which the virtual machine software is implemented." Perhaps the class of "foreigner-virtualizing" systems should be added to Goldberg's two classes that were alluded to earlier. Whatever words are used to express the close relationship between emulators and virtual machines, the relationship itself must always be kept in mind.

BIBLIOGRAPHY

1. Lichstein, H. A. When should you emulate?, Datamation 15, 11 (November 1969), 205-210.

2. Goldberg, R. P. Virtual machines - semantics and examples. IEEE Comput. Soc. Conf., Boston, Mass. (September, 1971), 141-142.

3. Goldberg, R. P. Hardware requirements for virtual machine systems. HICSS-4, Hawaii International Conference on System Sciences. Honolulu, Hawaii (January, 1971).

4. Mallach, E. G. Emulation: a survey. Honeywell Computer Journal 6, 4 (1972), 287-297.

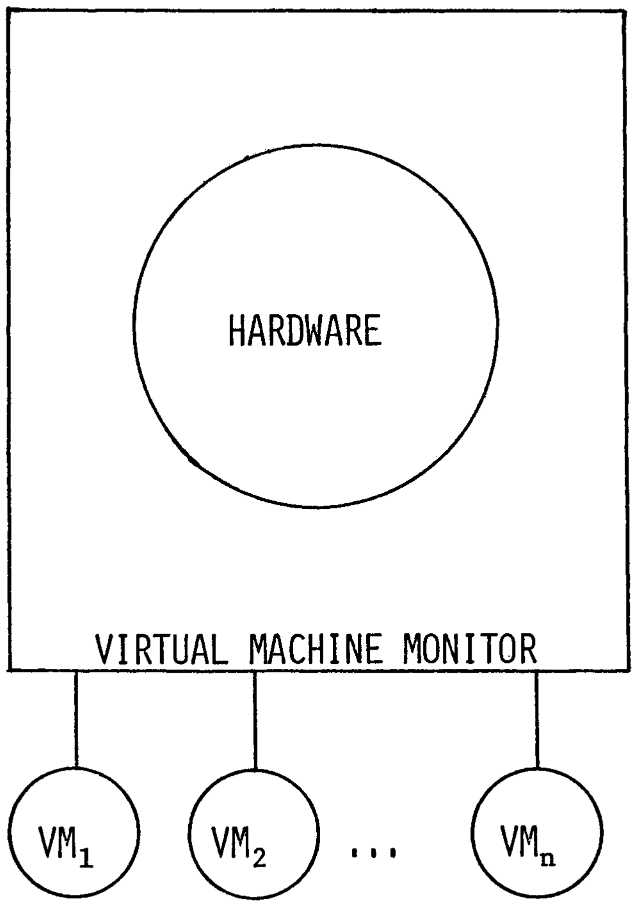
5. Allred, G. System/370 integrated emulation under OS and DOS. Proc. AFIPS 1971 Spring Joint Computer Conference, Vol. 39, 163-168.

6. Rosin, R. F., Freider, G., and Eckhouse, R. H., Jr. An environment for research in microprogramming and emulation. Comm. ACM 15, 8 (August, 1972), 748-760.

7. Srodawa, R. J. and Bates, L. E. An efficient virtual machine implementation. Proceedings of ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Harvard University, Cambridge, Mass. (March 26-27, 1973).

8. Fuchi, K., Tanaka, H., Manago, Y., and Yuba, T. A program simulator by partial interpretation. Second Symposium on Operating Systems Principles, Princeton, N. J. (October 1969).

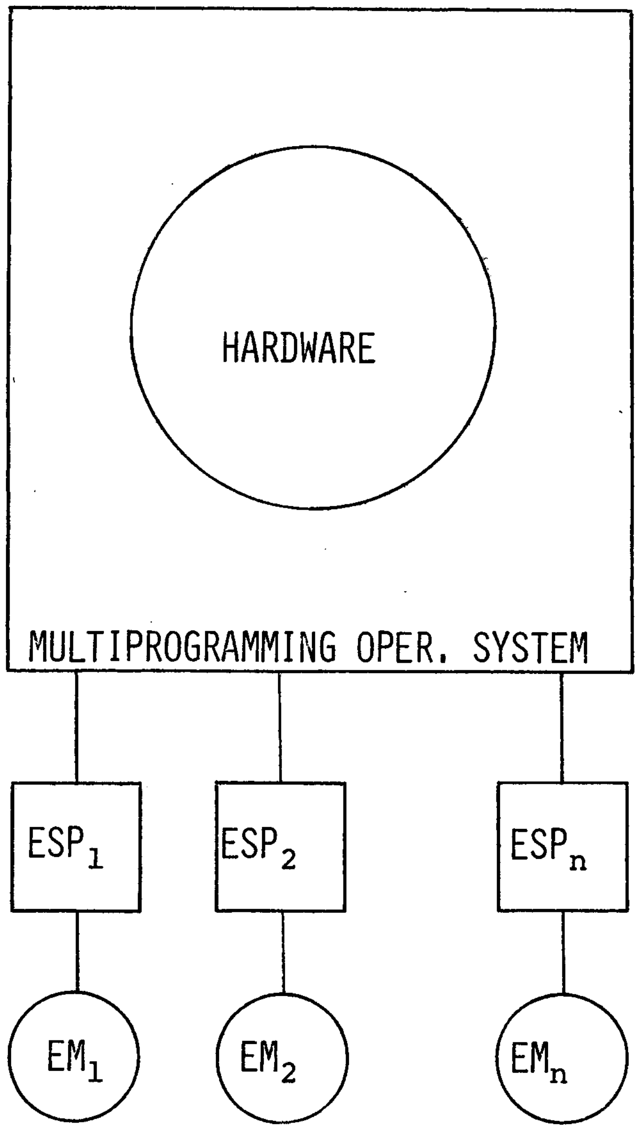
9. Galley, S. W. PDP-10 virtual machines. Proceedings of ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Harvard University, Cambridge, Mass. (March 26-27, 1973).



VM: VIRTUAL MACHINE

FIGURE 1

STRUCTURE OF AN ENVIRONMENT SUPPORTING MANY VIRTUAL MACHINES



ESP: EMULATOR SOFTWARE PACKAGE

EM: EMULATED MACHINE

FIGURE 2

STRUCTURE OF AN ENVIRONMENT SUPPORTING MANY EMULATED MACHINES