Narrowing the generation gap between virtual machines and minicomputers

by HENRY J. THEBERGE and ERIC E. BEAVERSTOCK
Honeywell Information Systems Inc.
Waltham, Massachusetts

INTRODUCTION

A virtual machine system has been defined by R. Goldberg\(^3\) as "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, . . .".

Large computer systems have been shown to be excellent environments on which virtual machine systems operate.\(^1,2\) If these systems were implemented on small computer systems ("minicomputers"), a desirable environment for software development would exist because of the capability of evaluating and testing different operating systems. Minicomputer applications such as graphics, process control, intelligent terminals, could be developed on a virtual machine while sharing support equipment. Current mini-computer inadequacies prevent the successful implementation of virtual machine systems. These inadequacies include:

- no automatic memory protection
- the lack of sensitive instruction recognition
- no virtual memory facilities
- no virtual I/O handling

This paper presents a method for implementing a virtual machine system in a mini-computer. The approach circumvents existing inadequacies by utilizing a software/firmware-implementable, interpretive, high-level, system programming language. The system, if implemented in software, has an emulator rather than a virtual machine monitor as the term has been currently defined.\(^5\)

This paper is divided into three parts

- Description of the higher level language
- Description of the Emulator
- Description of the Virtual Machine Monitor

DESCRIPTION OF THE HIGHER LEVEL LANGUAGE

The language we considered is TRAIL.\(^4\) The basic elements of TRAIL are:

1. a source language
2. an intermediate interpreted language
3. an interpreter

Source language

The source language was designed to facilitate system design and debug phases, to enhance documentation, and to allow greater productivity of programming. The language facilitates recursive programming and elicits a programming discipline for the separation of logical flow of control from the basic manipulative operation. TRAIL effectivity separates conditional tests and branching decisions (rules) from straight-line code (action sequences). Each program is called a graph and is comprised of rules and action sequences. An action sequence is a free statement consisting of a sequence of action calls, variable references, and action operators without any Go To or conditional instructions. Action sequences apply to a run time work-stack and may be nested to any depth. The work-stack is used mainly for expression evaluation and parameter passing.

Variables are allocated dynamically at graph-call time and deallocated at graph-exit time. Variables can access virtual addresses since TRAIL allows definition of a virtual memory mapping scheme.

These features provide greater productivity in design and debug phases, and enhance communication between programmers via simplified documentation procedures.

Intermediate interpreted language

The source language is converted into an intermediate or target language by a translation process. The target language is interpreted at execution time. This interpretive approach achieves portability, and programs are treated as relocatable data by the interpreter. Each action sequence is encoded into a string of byte sized intermediate language operators for interpretation. This choice of an interpreted target language accomplishes minimum object code size (roughly 50 percent smaller than assembled code) and machine independence of the developed software.

Interpreter

The interpreter is either written in assembly language or firmware implemented. It is designed to maximize speed
with respect to interpreting the byte sized string of intermediate language operators, and at the same time minimizes the actual code size required. When a graph is entered, the interpreter allocates relative space for an environment. The environment consists of the work stack and an area for the interpreter to maintain pertinent run-time information. By associating a unique environment with each graph or groups of graphs, the concept of coroutine is introduced. A coroutine is a graph which can relinquish control to another coroutine and later be reactivated to continue computation. All environment management is controlled by the interpreter. The interpreter I/O mechanism is activated, when an I/O request is identified. The mechanism performs a type of subroutine call to the I/O controller, which performs the desired I/O action.

These features of TRAIL along with others [4], provide an excellent base for the creation and support of an emulator or a virtual machine monitor for a small machine.

DESCRIPTION OF THE EMULATOR

Figure 1 depicts an emulator system through the use of language primitives as designed by Robert P. Goldberg of HIS.1 The architectural design consists of a basic machine that executes an emulator. The emulator is written in assembly code and performs all I/O operations, memory mapping functions, and other virtual machine requirements. The interpreters, also written in assembly code, interpret the TRAIL code of the various operating systems and user programs. An interpreter is executed directly by the basic machine and the interpreter passes control to the emulator after having interpreted an action operator requiring some virtual machine operation i.e. space allocation. The emulator, having performed the necessary operations, determines which interpreter is to be activated. The interpreter transfers control between the operating system and the user programs by means of the coroutine mechanism as previously described.

The emulator as proposed here, solves many problems that currently exist in implementing virtual machine concepts on mini-computers. Automatic memory protection is provided by the emulator because it controls the memory allocated to each interpreter and its programs. The interpreter(s) recognizes certain action operators to be sensitive—i.e., an instruction requiring some virtual machine function—thus the sensitive instruction is dealt with at the interpretive level. A virtual memory scheme is implemented within the emulator, and is activated by the emulator when required. Virtual I/O handling is also provided by the emulator.

Thus, with some software design modification, one can easily move toward emulation on a mini-computer.

DESCRIPTION OF THE VIRTUAL MACHINE MONITOR

Figure 2 depicts a virtual machine monitor system. The basic model requires that an interpreter be firmware (micro-coded) implemented. The virtual machine monitor is written in the language being interpreted (e.g. TRAIL). The operating system, also written in the same interpreted language, controls the execution of the various user programs. The
Narrowing the Generation Gap Between Virtual Machines and Mini-Computers

firmware transfers control between the operating system, user programs, and the virtual machine monitor by means of a coroutine mechanism. The firmware passes control to the virtual machine monitor when virtual machine functions are required. An important characteristic is that the virtual machine monitor is not performing an instruction by instruction interpretation of the operating system or user programs, but rather remains inactive until called upon by the firmware.

The virtual machine monitor accomplishes memory protection by gaining control from the firmware as required by the various programs. Certain action operators which require virtual machine functions are recognized and the virtual machine monitor is activated by the firmware. Virtual memory, is also handled by the virtual machine monitor when given control through the firmware. The virtual machine monitor accomplishes the mapping of the set of resources found in the virtual machine configuration into the set of resources existing in the basic machine.

This virtual machine concept provides for the creation of various extended machines upon which user programs may run. Since the virtual machine monitor is written in TRAIL, monitoring and debugging is readily accomplished.

CONCLUSIONS

From the described configurations, the problem of implementing a virtual machine system on small machines is reduced. The emulator approach enhances current capabilities of mini-computers without major modifications or extensions; however, a slow down in execution time will result. This problem can be avoided if the virtual machine monitor approach is taken.

The simplicity and compactness of the TRAIL language makes it possible to implement the virtual machine requirements on a mini-computer.

Although TRAIL was mentioned as the interpretive language to be used, it is noted that any interpretive language with the same basic characteristics can be utilized. With either of these approaches, a broader use and greater reduction of the limitations of mini-computers are accomplished.

BIBLIOGRAPHY


