The Virtual Resource Manager

Thomas G. Lang, Mark S. Greenberg, and Charles H. Sauer

Introduction

The Virtual Resource Manager, or VRM, is a software package that provides a high-level operating system environment. The VRM was designed to build upon a hardware base consisting of a Reduced Instruction Set Computer (RISC) and a PC AT compatible I/O channel, although it is not limited to this environment.[1] In fact, the VRM can be easily extended to support different I/O hardware. An example of this is the VRM's support of the IBM 5080 graphics hardware, which is designed to an IBM System/370 channel interface.

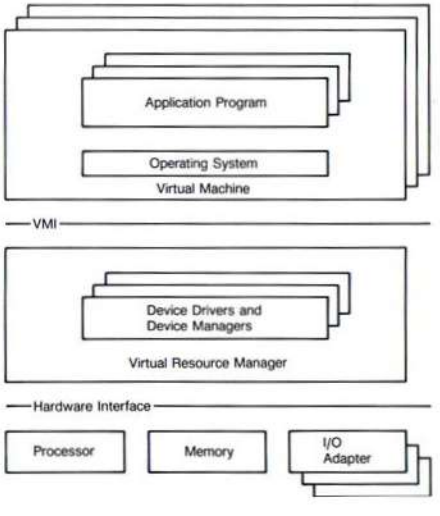

Figure 1 RT PC Software Design

The concept of RISC architecture is the minimization of function in hardware, providing only a limited set of primitives.[2] This allows the processor to be designed with simplified logic and a corresponding increase in the speed of its instruction set. In this environment, the software must provide function that traditionally is provided in hardware, such as integer multiply and divide functions and character string manipulation. The VRM builds on this hardware base to:

- Provide a high-level machine interface, which simplifies the development and implementation of operating systems and their applications.

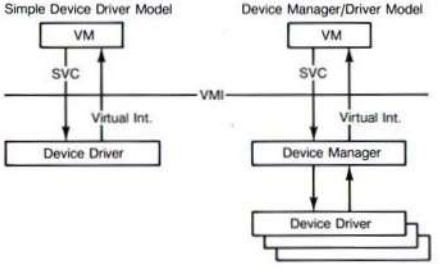

Figure 2 Virtual Device Models

- Maximize performance to support real-time process control type applications.
- Allow users to easily customize the system to meet their needs by providing an extendable, flexible interface.
- Provide compatibility with IBM-PC applications by supporting an Intel 80286 coprocessor.

The approach used to accomplish these goals was to design a Virtual Machine Interface, or VMI, with a set of functions to facilitate the use of a variety of operating systems. The VMI has features that support concurrency of multiple operating systems and applications, while insulating them from most details of the implementation of the hardware, except for the problem state instruction set. Also, the VMI allows operating system programmers to install extensions to the VRM to support additional I/O devices, or even to replace the IBM-supplied I/O subsystems.

Traditionally, virtual machine implementations have suffered in performance due to the overhead of simulating hardware function. The key to maximizing the performance of the VRM is that the vast majority of instructions issued by the operating systems and applications are directly executed by the hardware. The VRM software is invoked mainly to handle I/O operations at a relatively high functional level.

Fundamental to the design of the RT PC is that the VRM is the underlying support layer for an operating system. In particular, the UNIX kernel[6] was chosen as the principal operating environment supported by the RT PC product, and the design of the VRM was influenced by this selection.

The concept of a virtual machine has been implemented on IBM mainframe computers with a software product known as VM/370.[3] The VRM is similar to VM/370 in that it supports the concurrent execution of multiple operating systems. However, there is a significant difference. VM/370 provides a complete functional simulation of the real System/370 hardware, such that an operating system built for the real hardware, like MVS, can run in a virtual machine. The Virtual Machine Interface supported by the VRM provides considerably more function than the RT PC hardware; an operating system implemented to the VMI will not run on the real hardware. The design of the VMI traded off complete hardware compatibility for the benefits of a high-level, high-function machine definition.

Along with the concept of concurrent virtual machines, the VRM supports virtual memory.[4] The hardware memory management capabilities include a 24-bit address space for real memory (i.e., the ability to address up to 16 megabytes of real memory) and a 40-bit address space for virtual memory (1024 gigabytes, or one terabyte).[5] The virtual address space is comprised of 4096 segments of 256 megabytes each. Sixteen segment registers are provided by the hardware, and one of them is permanently dedicated to addressing I/O devices. Thus, up to 15 segments can be accessed simultaneously. The VRM software takes advantage of these features to logically separate the address spaces of the virtual machines from each other and from the VRM address space.

Another VRM feature, related to virtual memory, is "mapped file" support. Mapped files are a relation of logical disk blocks to virtual memory addresses, such that a disk file can be read from or written to simply by reading from or writing to its associated memory addresses. Explicit disk reading and writing is not required.

The AIX operating system contains a complete file system, based on explicit disk reading and writing. When modifying AIX for the VMI, it was desirable to salvage as much software as possible. Also, the concept of mapping files does not work well with removable media, such as tapes or diskettes. So, mapped file support is augmented with a minidisk manager in the VRM, providing more conventional file access support.

The minidisk manager provides access to disks partitioned into separate spaces, or minidisks. In turn, the minidisks are partitioned into logical blocks whose size is determined by the operating system, independent of the characteristics of the physical disk. The minidisk manager also includes functions not normally found in simple hardware access methods, such as error recovery and bad block relocation. Further, the VMI for the minidisk manager allows the potential for "remote" minidisks, accessed across a communication link such as a high speed local area network.

The "virtual resource" concept is also applied by the VRM to I/O devices, such as virtual terminals.[7] The VMI includes a high-level interface to I/O devices that is consistent for all devices. Also, the VMI includes provisions for bypassing the VRM and accessing devices directly. The preferred method of using a device from a virtual machine is to take advantage of the I/O support functions supplied by the VRM. But, there are graphics applications, for example, which can gain enough performance by writing directly to a display device to offset the loss of flexibility suffered when bypassing the VRM services. Another reason for allowing direct access to I/O devices was compatibility with existing applications; for example, a BASIC language program written using the PEEK and POKE functions to access an I/O device.

Extendable Virtual Machine Architecture

Another feature that distinguishes the VRM is the extendability of the architecture. Users of microcomputers have become accustomed to plugging new devices into a machine's I/O channel. However, getting the machine's software to use the device usually requires some ingenuity. One approach is to design the new device such that it "looks like" an existing device, so that the existing software can recognize and use it. Another approach is to run an application program that drives the device directly, independent of the existing operating system. For example, a program could communicate with a device by sending commands to its I/O port, then using a software "spin loop" to poll its status port to determine when the commands complete. The former approach limits the flexibility of the new device, while the latter destroys the effectiveness of a multiprogramming operating system by tying up the processor during I/O operations.

The VRM allows a new approach, whereby software for a new device can be fully

integrated into the existing operating system. Further, the reconfiguration of the VRM to add or replace software can be performed in real time without disrupting the normal function of the machine.

A data structure, known as a Define Device Structure, or DDS, is included in the VMI so that a programmer can describe the attributes of a new device and its related software support to the VRM. Information in the DDS includes the I/O port address(es) used by the device, which channel interrupt level it uses, which DMA channel it uses (if any), whether it has any resident RAM or ROM, etc. Also, the DDS indicates which program module should be called to process such functions as:

- Device initialization
- Interrupt handling
- I/O initiation
- Timeout or exception handling
- Device termination

Using information from the DDS, the VRM is able to determine which user-installed program to call to handle an interrupt generated by an installed device. The additional software required to support a new device is added in real time, in contrast to existing systems that require the use of an off-line or stand-alone program to reconfigure the system.

To use devices, the VMI contains a set of functions including:

Define Code Install software into the VRM, or delete installed software.

Define Device Install a DDS into the VRM, or delete an installed DDS.

Attach Reserve a device and allocate any resources its software may require.

Detach Undo the function of "Attach."

Send Command Send a command to a device.

Start I/O A variation on "Send Command," which allows a set of commands, or buffers, to be sent to a device.

To use a device, a logical connection ("path") is established between the user and the device. The Attach function is used to establish a path, and a path identifier token is returned to the user. Subsequently, the path identifier is used to send requests to the device. When the device completes the request, it returns status information or an interrupt to the user, using the path identifier to route the data.

The VMI defines two ways to send requests to a device, the Send Command and Start I/O functions. Parameters for these functions include the path identifier in addition to device specific parameters such as a request code and buffer pointer. The difference between the two functions is that the latter passes its parameters in a data structure, know as a Channel Control Block, or CCB, which allows the specification of a chain of commands or buffer pointers. This can be useful, for example, when using a device that supports "scatter/gather" functions. During a read request data can be input from a device and "scattered" into different memory buffers. Or, during a write request data can be "gathered" from different buffers and output to a device.

Another parameter for the two request functions is an operation option that determines if the request is to be processed by the VRM synchronously or asynchronously. Implicitly, this also determines how completion status is returned to the virtual machine. For synchronous requests, completion status is supplied as a return code from the requested function, while the completion of an asynchronous request is indicated by a "virtual interrupt". The VMI defines nine interrupt levels for a virtual machine, which allows the assignment of relative priorities to interrupting conditions. When not processing an interrupt, the virtual machine is considered to be on level 7. Seven levels can be assigned to interrupting I/O devices. In order of decreasing priority, they are levels 0 through 6. In addition, there are two other levels. The machine communications level is used for messages between the VRM and the virtual machine. The highest priority level is the program check level, which is used by the VRM to report exception or error conditions to the virtual machine. The return code from a synchronous request provides 32 bits of status, while up to 20 bytes of status can be supplied with each virtual interrupt.

Two types of programs can be installed into the VRM: device drivers and device managers. A device driver is a collection of subroutines that support a specific hardware device. The VRM synchronously calls the subroutines to handle device-specific functions, such as handling interrupts and time-out conditions, and processing I/O commands from virtual machines. The VRM device driver support is intended to be

sufficient for implementing relatively simple devices, such as printers, diskette drivers, and tape drives.

Device managers provide an additional level of support for more sophisticated devices, such as virtual terminals or communications subsystems (see Figure 2). These types of device subsystems typically have requirements to handle multiple asynchronous events and to manage different types of resources. For example, the Virtual Terminal Manager coordinates the activities of device drivers for the keyboard, display, speaker, and locator to simulate a higher level device known as a "terminal."

Allocation of System Resources

Resources in the VRM are categorized as serially reusable or shared. Serially reusable resources are those that can be used by different applications, but only by one at a time. For example, multiple applications may use the printer but one application must finish before the next takes over. Otherwise, the result would be scrambled printer output. Shared resources, though, may be used "simultaneously." Examples include the disks and memory, which are shared by dividing them into logical pieces (minidisks and segments), and the processor and communication lines, which are shared on a time basis.

The VRM manages several shared devices, most notably the keyboard, locator, speaker, display, and hard files. Virtual machines can have many logical terminals. The user controls which logical terminal is associated with the physical hardware via a set of reserved key sequences. Virtual terminal input is routed by the VRM to the owner of the screen that has been selected for display by the user. Output to virtual terminals is updated in memory if that display is not selected.

Support of the PC AT coprocessor presented some interesting challenges for resource management.[9] The main constraint was that the VRM had to be transparent to the applications using the coprocessor. A considerable amount of hardware support is dedicated to this purpose, in the form of "trap" logic that monitors access by the coprocessor of I/O addresses.[8] For nonshared devices, the VRM reserves the device for exclusive use by the coprocessor. I/O operations using devices of this type proceed with no further intervention required by the VRM. When using shared devices, however, the VRM must intercept each I/O operation requested by the coprocessor and simulate the function as if it were dedicated to the coprocessor. For example, when the coprocessor writes data to what it thinks is the display screen, the VRM saves this data in a memory buffer. And, when the coprocessor's virtual terminal becomes the "active" terminal, the data is moved to the actual display buffer. Also, at this time keystrokes are routed to the coprocessor when it accesses what it thinks is the keyboard adapter's I/O port. Notice that since the coprocessor accesses nonshared devices directly, they perform at precisely the same speed as they do in a PC AT. However, shared devices suffer some performance penalty since functions must be simulated by the VRM software.

Another resource that can be shared with the coprocessor is memory. The VRM can reserve some of its own memory for use by the coprocessor. In this mode, memory translation hardware detects memory references by the coprocessor and routes them to the VRM's memory. Alternatively, a memory card can be plugged into the I/O channel, and coprocessor memory references will be directed to it. This allows a great deal of flexibility to trade off the lower cost of shared memory against the higher performance of dedicated memory. The trade off is not "all or nothing." For example, a 1-megabyte address space can be provided for the coprocessor using a 512-kilobyte memory card and sharing 512 K of system memory.

Virtual memory is utilized by the VRM to eliminate arbitrary restrictions on resource usage. It is not uncommon for operating systems to restrict the number of processes in the system or the number of devices that are supported. The VRM defines internal control block areas in virtual memory that are large enough to support thousands of processes and device drivers. Thus, limitations are a function of the amount of real memory, disk space, and I/O channel slots available on a particular machine.

Designs for Real-Time Performance

The VRM was customized for the real-time processing environment, compensating for the shortcomings of the kernel in this area. Features of this design include:

- Low overhead creation and deletion of new processes and interrupt handlers
- Efficient interprocess communication
- Preemptable processes and interrupt handlers, to minimize interrupt latency time
- Prioritized scheduling of processes and interrupt handlers

- Interval timer support with 1-millisecond granularity.

Multi-programming is implemented in the VRM by dividing work into logical units, or "processes," which are scheduled by priority. In addition, the VRM contains "interrupt handlers," which are invoked in response to interrupt signals from hardware devices. In "Extendable Virtual Machine Architecture" on page 120, programs in the VRM were characterized as device managers or device drivers. Device managers, and virtual machines, are represented as processes in the VRM, while interrupt handlers are among the subroutines that comprise a device driver.

Processes and interrupt handlers can communicate using shared memory, or by using the VRM's interprocess communication functions, which include queues (for message passing) and semaphores (for serialization and synchronization).

Particular emphasis was placed on supporting high-speed devices, with stringent latency time requirements. Hardware interrupt processing is the highest priority work in the system. Interrupts from devices are further divided into four priority classes, such that the servicing of an interrupt can be preempted by a higher priority interrupt.

Also, an "off-level" interrupt handler capability is available that allows a device interrupt handler to process time-critical operations without being preempted, and to defer less critical processing to a lower priority level that can be preempted by other device interrupts.

After all pending interrupts are handled, the VRM selects the next process to execute based on 16 priority levels. The selected process will remain executing until it "waits" for some condition (such as the completion of an I/O operation or the arrival of a new work request), or until it is interrupted. Among processes with the same priority, "time slicing" is implemented; that is, if a process does not relinquish control after a period of time, the VRM will suspend it and pass control to another process. The default time slice interval is 16 milliseconds, and this value may be increased in increments of 16 milliseconds. If a sufficiently large increment is selected, time slicing is effectively disabled.

The design of the data structures for multiprogramming was influenced by performance considerations. The processor has a large number of registers (16 system registers, 16 segment registers, and 16 general-purpose registers), which makes context switching between applications a lengthy job. In a typical operating system, when an interrupt occurs, the state of the interrupted program is saved in a known location, then transferred to a control block associated with the interrupted program if it becomes necessary to switch control of the processor to a different program. In the VRM, this would require moving a large amount of data, so the interrupt handlers are set up such that the state of an interrupted program is saved directly into its control block. This contributes to faster context switching.

Another aspect of the control block design that contributes to fast context switching is that the "dispatcher," which selects which program next gets control of the processor, never has to search through queues of control blocks. The control blocks for programs that are ready to execute are always kept sorted by priority, thus only about 1% of the total time required for a context switch is required to select the next program. The remaining time is spent saving the state of the current program and restoring the state of the next program.

The VRM was designed using top-down structured programming techniques. The program code was written first in a high-level language, using primarily PL.8 (an internal IBM development language, derived from PL/I).[10, 11, 12] After the system was functioning to the point where meaningful applications could be implemented and run, the performance of the system was measured in detail. The performance data was used to determine critical paths in the software, or "bottlenecks." These parts of the system were then tuned to maximize performance. The first step in tuning was to attempt to make the PL.8 code more efficient. In many cases, this tuning turned out to be sufficient to meet performance objectives. However, some critical paths required recoding in assembler language to achieve desired performance.

The process of tuning the system was an iterative one for the measurement and recoding steps. For example, one performance objective was that the disk device driver be able to handle a disk formatted with a 2:1 interleave factor without missing revolutions, with enough of a margin to allow for an interrupt from an Async communications adapter during the critical path. A factor that increased the difficulty in meeting this objective was the disk hardware, which does not support DMA for transferring data between the adapter and memory. The disk hardware, chosen mainly on cost and compatibility considerations, is similar to the PC AT disk hardware. Using that hardware, the PC AT supports a 3:1 interleave.

In pursuit of the 2:1 objective, the VRM interrupt handling logic and disk device driver were measured and recoded numerous times, each time squeezing out a few more microseconds from the path length, until the objective was met. At several stages in the process, software ingenuity was required to surmount hardware timing limits. Some of these software "tricks" included:

- Sorting the queue of disk requests according to sector/track number, influenced by the current position of the disk arm
- Looking ahead in the queue when one request completed, to anticipate the requirements of subsequent requests
- Sending the next command to the disk adapter before processing of the current command is complete
- Using a table look-up algorithm to determine how long a "seek" operation should take, based on current and future arm position, then setting a timer to wake up the disk driver just prior to the operation completing
- Taking full advantage of the overlapped load, store, and branch capabilities of the pipelined processor.

In this extreme example, the large tuning effort paid off when a difficult objective was met. Fortunately, most other tuning problems were easier to solve. Also, there were "spin-off" benefits gained in the disk driver tuning. The path length reductions in the VRM common interrupt handling logic benefitted all device drivers, and some of the techniques used in the disk driver were applied to other device drivers. In particular, the overlapped processing of queued requests increases the throughput of all devices.

Critical to the job of performance tuning was accurate measurement of the system. Three different techniques were used. First, selected operations were executed repetitively, so that elapsed time could be measured. The measurement device was a stop watch, so to eliminate reaction-time errors and to increase accuracy, the repetition factors were chosen to be very large (e.g., thousands or even millions of iterations). Some of these "bench mark" loops were internally developed, while others were selected from bench marks published in trade journals. The latter type of bench mark was especially useful when comparing performance of competing systems.

The second type of measurement was done by inserting "hooks" into critical paths. These hooks consist of I/O instructions that output data to reserved channel addresses. To obtain measurements, a special I/O adapter is plugged into the channel to monitor the output from the hooks. The data collected by this adapter is saved on a tape. Afterwards, the tape is input to a data reduction program that generates a path flow analysis with timings. This technique allows very sophisticated path analysis, but suffers the drawback that the hooks themselves take a small amount of time to execute. Although the hook execution time is relatively small, the cumulative times can, in some cases, add up to a significant amount. Also, as the interval between hooks decreases, the hook's execution time becomes proportionately more significant.

The third technique involved a logic analyzer to monitor the output of signals from the processor chip. Using the analyzer, it is possible to measure precisely the time it takes to execute individual instructions or sequences of instructions. This is impractical for measuring large programs, but is well suited for analyzing small sections of program code that are executed very frequently. For example, program context switching and interrupt handling functions execute hundreds of times per second. In these critical paths, a few microseconds can be significant.

A great deal of performance tuning effort was spent maximizing the "pipeline" effects of the ROMP processor. The pipeline effects result from the processor's ability to overlap various stages of instruction execution. Two different situations illustrate these effects. First, if the next instruction(s) after a memory load instruction do not use the value being loaded from memory, they may be executed in parallel with the memory access. By properly interleaving instructions, this effect can be exploited to reduce the total execution time of a sequence of instructions. Second, when a branch instruction is executed, the processor must reload its instruction pre-fetch buffer with the new instruction stream. By using the processor's Branch-with-Execute instructions, it is possible to overlap the execution of one instruction with the pre-fetch buffer reload time.

The high level language compilers for the RT PC, in particular the PL.8 compiler, are designed to take advantage of the pipeline effects of the processor. For assembler programmers, the pipeline effects can be utilized, although usually at the cost of cleanly structured programming. For the tightly optimized critical paths in the VRM this has

been done, but the programming effort required, contrasted with the high efficiency of the compilers, has resulted in the majority of the VRM being implemented in high-level language.

Conclusions

The VRM builds upon the low level RT PC hardware interface to provide a high-function system environment. It brings to a desk-top microcomputer many features that formerly were found only on much larger, more expensive systems, such as virtual memory and virtual I/O subsystems. It also includes features, such as dynamic reconfiguration and an extendable architecture, which are unique; and it allows for the migration of existing UNIX and IBM PC based applications to a new architecture.

During the past several years of development, the RT PC hardware underwent several major changes, but the Virtual Machine Interface has remained relatively stable throughout this time, thus minimizing the impact of the hardware changes to the implementation of AIX and its applications.

The VRM's functions complement the hardware instruction set, providing features such as virtual memory, virtual devices, minidisks, and multi-programming. This creates an environment for implementing operating system extensions and hardware device support that has the flexibility to evolve as the hardware technology evolves without forcing radical changes to existing software.

Acknowledgments

The authors would like to acknowledge the efforts of the people who contributed to the development of the VRM. The virtual terminal software was developed by Lynn Rowell's department. The RAS and Install were developed by Hira Advani's department. The VRM device drivers were developed by Mark Wieland's department. Special thanks go to Joe Corso and each member of his department thoughout the VRM development for the VRM design, the testing methodology, and the technical leadership for integration of the product.

References

1. George Radin, "The 801 Minicomputer," *IBM Journal of Research and Development,* 27, pp. 237-246, May 1983.
2. D.A. Patterson and C.H. Sequin, "RISC: A Reduced Instruction Set Computer," *Proc. 8th Annual Symposium on Computer Architecture,* May 1981.
3. R.A. Meyer and L.H. Seawright, "A Virtual Machine Time-sharing System," *IBM Journal of Research and Development,* Volume 9 Number 3, 1970.
4. J.C. O'Quin, J.T. O'Quin, Mark D. Rogers, T.A. Smith, "Design of the IBM RT PC Virtual Memory Manager," *IBM RT Personal Computer Technology,* p. 126.
5. P.D. Hester, Richard O. Simpson, Albert Chang "IBM RT PC ROMP and Memory Management Unit Architecture," *IBM RT Personal Computer Technology,* p. 48.
6. Larry Loucks, "IBM RT PC AIX Kernel — Modifications and Extensions *IBM RT Personal Computer Technology,* p. 96.
7. D.C. Baker, G.A. Flurry, and K.D. Nguyen, "Implementation of a Virtual Terminal Subsystem," *IBM RT Personal Computer Technology,* p. 134.
8. John W. Irwin, "Use of a Coprocessor for Emulating the PC AT," *IBM RT Personal Computer Technology,* p. 137.
9. Rajan Krishnamurty and Terry Mothersole, "Coprocessor Software Support," *IBM RT Personal Computer Technology,* p. 142.
10. M. Auslander, et al., "An Overview of the PL.8 Compiler," ACM, 0-89791-074-5/82/006/0022.
11. Alan MacKay and Ahmed Chibib, "Software Development Tools for ROMP," *IBM RT Personal Computer Technology,* p. 72.
12. M.E. Hopkins, "Compiling for the RT PC ROMP," *IBM RT Personal Computer Technology,* p. 76.