The growing use of the virtual machine concept has resulted in the necessity for communication between the virtual machines. The design and operation of the Virtual Machine Communication Facility is discussed as an approach to offering such communication. The facility is an interface allowing a logical connection between two or more virtual machines. Potential applications for this facility conclude the discussion.

A formal approach for communication between logically isolated virtual machines

by R. M. Jensen

A virtual machine (VM) is a well-defined hardware architecture that is presented to a terminal user in a somewhat illusive manner. For those users who understand the concept, it is like having an entire real computing system at their disposal. To most CMS (Conversational Monitor System) users, the system is a time-sharing and program development tool, and they give little thought to the actual machine capabilities that exist. The virtual machine is an assimilation of its counterpart, the real machine, yet its users and requirements may vary widely.

The requirements for communication between virtual machines may be the same as for real machines, or may be closely related to the requirements of inter-CPU task communication. This paper discusses the Virtual Machine Communication Facility (VMCF)[1,2] as designed and implemented for Virtual Machine Facility/370 (VM/370) Release 3, Program Level Change (PLC) 8 and subsequent releases. VMCF is a software interface that provides a logical connection between two or more otherwise isolated virtual machines. The interface is not an application but is rather a facility upon which applications may be built. Presently, the implementation does not include a high-level language or macros; instead, the execution of functions is achieved directly through use of the VM/370 diagnose interface (conceptually similar to SVC-level programming in other operating systems). The first part of

this paper provides a rationale for developing VMCF and discusses advantages and disadvantages of prior methods for intermachine communication. The other methods are discussed purely from a virtual machine point of view; any applicable benefits for native operation are not included.

The next part of this paper presents an overview of the design and functions provided by VMCF. The discussion here may appear to be at a more detailed level than is necessary for this paper; however, the components discussed are the externals associated with the interface. This is true partly because of the nature of a virtual machine itself. A user, if need be, is not bound by operating system software but, in fact, has the almost limitless flexibility of machine languages and hardware. VMCF is an interface at that level.

Next, the paper discusses the types of potential application power that exist using VMCF and/or intervirtual machine communication in general. Some of the examples cited are hypothetical, whereas others are actual implementation. The purpose of this section is not to discuss specific applications themselves but rather the types of applications that could be developed.

Last but not least, a synopsis of VMCF is provided with concluding remarks to support the paper itself.

Background

The idea of communication between logically isolated virtual machines is not new. Many methods have been employed by both IBM and users and can be traced back to the early days of CP-67, the predecessor of VM/370. The inherent design of the control program (CP) provides a capability to transfer spool files from one virtual machine to another. A virtual (simulated) channel-to-channel adapter interface (CTCA) was developed to support the testing of a loosely coupled multiprocessing environment in a virtual machine (OS/ASP—Operating System/Attached Support Processor).[3] In a few cases, hardware communication lines were used to link together one or more virtual machines (primarily for testing purposes). In spite of existing facilities supported by the control program, several other methods were employed (requiring CP modifications) to provide a more efficient and general-purpose interface. These included VMCF-like functions and/or MP/AP (multiprocessor/attached processor) simulation using the SIGP (signal processor) hardware instruction. VMCF, itself, is an evolution of prior modifications to the control program.

The use of spool files for communication is desirable for certain classes of data but is not efficient as a transaction and/or message

process involving storage-to-storage data transfer. The spool file data itself must first be placed on a real spooling device (DASD—direct access storage device) before it can be transmitted to another virtual machine. The data then must be read from the spooling device and is generally implemented as a simulated I/O interface (virtual unit record I/O). The advantage of this interface is that it approximates a store-and-forward technique, and the data is checkpointed and preserved across system failures. This type of communication is desirable, particularly for critical data files and network-like teleprocessing environments.

A possible virtual, simulated MP/AP environment implies the use of shared read/write storage and is designed primarily for control program access rather than for the application level. The use of such a facility for intermachine communication is beyond the scope of any known application, other than software development of a MP/AP hypervisor. The use of shared read/write storage could have its advantages (fast data move) but could be difficult to manage when a large number of virtual machines are involved.

virtual CTCA

The virtual channel-to-channel adapter appeared to be the most prevalent for certain applications involving intervirtual machine communication. The support was a standard part of the VM/370 system control program, and did not require user modifications. The interface provided by the CTCA is part of the architecture within the hardware and as such could be used without modification to other existing software (particularly the operating systems). Use of the virtual CTCA did not preclude potential support for native CPU communication. The simulation of the CTCA by the control program is consistent with its native operation and characteristics.[4] The simulated method for transferring data is from virtual storage to virtual storage and does not require internal buffering by the control program or excessive free storage demands on the control program. The virtual CTCA was not constrained by the availability of physical hardware and is totally software-simulated. Data is transferred using the MVCL and MVC instructions (read and read backward, respectively) and all hardware functions (e.g., storage protection and channel commands) are applied to the simulation. The method of communicating from storage to storage was clearly more efficient than other existing facilities.

The virtual CTCA serves several purposes well, particularly for ASP-like environments that require direct CPU-to-CPU communication and involve a small number of virtual machines. But the progression of virtual machine subsystems, central server applications, and network-like communication among a large number of virtual machines precludes use of the CTCA.

The CTCA was designed to communicate from one CPU to another and not from one CPU to many others. The general use implies

that two physical (or virtual) channels are connected together. The channels may belong to the same or separate virtual machines. The control program provides a couple command to connect the two virtual channels. The user of the couple command must understand the hardware (virtual) configuration of the target virtual machine, i.e., the virtual address of the CTCA. If the coupling (or connection) of virtual machines is to be dynamic, then there must be a mechanism to communicate the available CTCA addresses (assuming a multiaccess environment). The virtual CTCA is also a simulated hardware function and is constrained by normal hardware limitations, i.e., the number of channels and devices. The CTCA itself occupies an entire control unit position.

The CTCA I/O is an asynchronous process but does not use the block multiplexing capabilities of a System/370 channel; hence, it is difficult to provide for concurrent data transmissions on a single channel. The I/O interface is also not generally well understood at the application level. The implementation of virtual CTCA support by CP involves full CCW (channel command word) translation for both sender (source) and receiver (sink) channel programs. This implies that the data pages (4K blocks) required to complete the operation must be fixed (locked) in real storage, and, depending on the amount of data, could potentially impact the operation and performance of the system.

VMCF characteristics

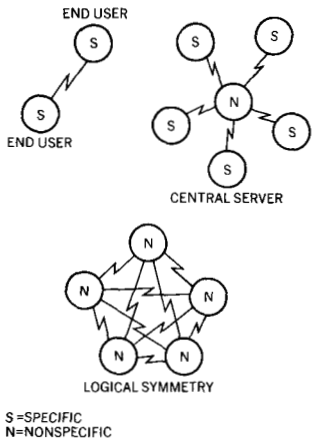
The design and implementation of VMCF, then, is a solution to a problem. The functions provided fit well for applications that require virtual machine communication and do not have dependencies on native CPU communication. The method employed is fast and efficient and overcomes many difficulties observed with prior methods of intermachine communication. VMCF provides simple and symmetrical protocols that can be understood at the application level. The connection of one virtual machine to another is a logical process for the duration of a single transaction. There is no master-slave relationship, and any authorized virtual machine can communicate with any other authorized virtual machine in a logically symmetrical fashion. The movement (transfer) of data is from virtual storage to virtual storage, and only one 4K page is ever fixed (locked) at a time during a single transaction (regardless of data size). The process is totally asynchronous (as with CTCA) but is not constrained by hardware limitations.

The inherent design of VMCF provides a queuing mechanism that allows multiple concurrent transactions to be processed by a single virtual machine(s). The transactions may originate from one other machine or many other machines (function of the application). Control of the access to a particular virtual machine is determined by its authorization state. A basic nonspecific state will enable a user (virtual machine) to accept requests from all other authorized users. A specific state implies that a user may

receive requests from one other user only. In either case the VMCF queuing structure allows multiple transactions to be stacked and processed by a given virtual machine within the constraints of its authorized state. Figure 1 shows three basic VMCF logical configurations based on the specific/nonspecific authorization mechanism:

1. The end-user to end-user configuration indicates that both virtual machines have authorized specific states for each other and will not accept VMCF requests from any other users. The application of this configuration would be most appropriate in a test environment. One virtual machine may be the target while the other is an externally controlled simulator (e.g., device simulator, error injector, terminal script).
2. The central server configuration implies that all users of the application have an authorized specific state with the server machine and will accept requests (transactions) from the server only. The server, on the other hand, is authorized as nonspecific and will accept requests from any user wishing to use the application. The application may involve a subsystem that provides system-wide services and/or a mechanism for sharing computing resources (e.g., data base, query, hardware devices, software).
3. The last configuration is the basic nonspecific state whereby all authorized users may be logically connected to each other. This configuration is appropriate for applications that involve multiple subsystems or central servers or where specific authorization is too restrictive. It should be noted that the VMCF authorized state is controlled by the user and may be dynamically changed at any time.

Figure 1 VMCF logical configurations



A disadvantage of VMCF is that it is not supported by other major system control programs (e.g., DOS/VS, VS1, VS2) and to do so would require modifications to the subject system control program. In viewing the design and architecture of VMCF, we can make an analogy with a channel-to-channel adapter, since many of the principles are similar.

Design and architecture

From a user point of view VMCF consists of the following basic components:

- A diagnose function (instruction) to initiate a VMCF request or subfunction. The hardware diagnose instruction is not supported in a virtual machine in the same manner as on a native CPU. Rather, the instruction is used to provide a special interface to the control program from a virtual machine. The VMCF diagnose function is one of many provided by the control pro-

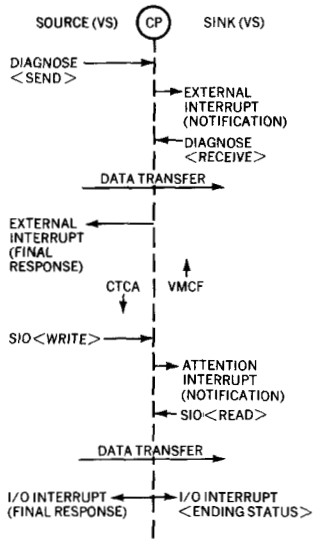
gram. The use of this function is similar to a SIO (Start I/O) instruction executed to a channel-to-channel adapter, i.e., it initiates a control or data transfer sequence. The diagnose function provides return codes indicating successful initiation of the request or error conditions associated with the request (e.g., SIO condition codes).

- A user parameter list that describes the request. The use of the parameter list is similar to a CTCA I/O channel program, i.e., it indicates the type of request (read-write-control) and provides data addresses and lengths associated with a data transfer operation. The parameter list additionally directs the request to a specific virtual machine (known by a user identifier, or USERID) and provides a message identifier to distinguish the request. The uniqueness of a VMCF request is distinguished by both the target user identifier and message identifier. A user, for example, may selectively choose a message (data) sent to his virtual machine based on the source user and message identifiers.
- Simulated hardware external interrupts to synchronize a request. The purpose of the external interrupts is twofold. A send external interrupt is used to notify the receiver (sink) of a sender (source) request. This interrupt is equivalent to an attention interrupt presented by a channel-to-channel adapter. As a response an external interrupt is generated for the sender to signal completion of a request. This interrupt is equivalent to an ending status I/O interrupt for a CTCA I/O operation. The rules for transferring data with VMCF are identical to those in the I/O system. The buffers used to contain data cannot be reused until the operation has been completed (the VMCF response external interrupt). Similarly, the same storage protection (store-fetch) and addressing considerations apply equally to the VMCF request.
- A message header describing the external interrupt. The architecturally specified hardware data provided with an external interrupt is not enough to describe a source request or completion of a request. VMCF, then, reflects additional data with all external interrupts, i.e., the message header. The message header data is stored in a predefined virtual storage location that is specified by a virtual machine when it authorizes (separate diagnose subfunction) a VMCF communication. The message header describes the source request to the receiver virtual machine and is in essence a copy of the source parameter list. The USERID, however, is changed to indicate the source user or originating virtual machine. To the sender virtual machine the message header describes the completion status for the original request. The completion status serves the same function as ending channel status word status and/or sense data for an I/O operation. Also, the completion status provides residual data counts indicating how much data was transferred during the operation (equivalent to I/O channel

status word residual counts). In all cases the message header contains status flags indicating the type of external interrupt (send or response).
- Data

Figure 2 depicts an analogy between a typical VMCF request and a CTCA I/O operation. Several commonalities and distinctions are described below.

Figure 2 CTCA/VMCF protocol analogy



Both processes are asynchronous. The CTCA is attention-driven, and VMCF is external interrupt-driven.

Both processes are symmetrical. Either side may be source or sink without distinction, and the protocols are functionally balanced.

Both processes involve storage-to-storage data transfer and are architecturally consistent (by definition).

The CTCA I/O operation shown is not exactly typical. The attention interrupt presented to the sink machine is normally not enough to indicate the type of request (although possible with unidirectional protocol). In reality an additional I/O operation would be required to read the command byte of the source request, i.e., indicates type of request—read, write, control.

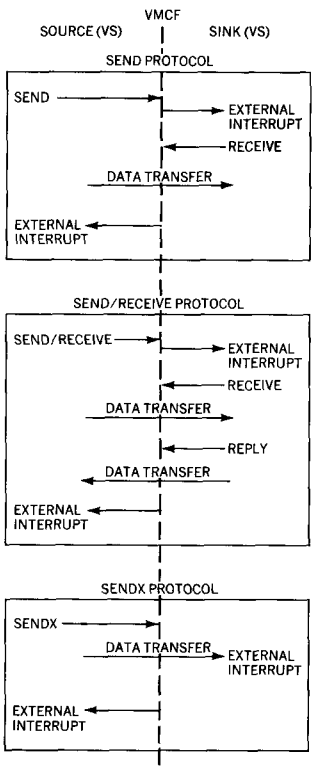
The CTCA I/O operation requires an ending status interrupt for the sink machine which is not necessary with the VMCF request (ending status-return codes occur when the diagnose instruction completes).

There are other architectural details that would tend to distinguish VMCF from the I/O system (e.g., VMCF data areas must be contiguous within a given transaction, whereas the CTCA will allow command-chaining/data-chaining into noncontiguous areas, and the CTCA protocol will allow data to be solicited with a read (truly symmetrical), whereas VMCF requests must be initiated with a write-type request, etc.).

protocols

A single protocol is all that is needed to communicate and transfer data between virtual machines. VMCF provides several protocols to assist in performance and usability and to enforce a formalization of the interface. The flow of data between virtual machines may be either unidirectional (source to sink) or bidirectional (source to sink to source). The enforcement of protocols is at the transaction level, that is, only during the transmission and/or exchange of data for a single request. VMCF communication may consist of a single protocol or combinations of all available protocols for multiple transactions. The connection of two virtual machines is a logical software process (by USERID) and is bound

Figure 3 VMCF protocols



only by those elements involved in a data transfer operation. Figure 3 portrays a relationship between the various VMCF protocols, which we now define.

The *send protocol* implies a one-way (unidirectional) transmission of data from the source machine's virtual storage to the sink machine's virtual storage. The send diagnose function is used by the source to initiate the transaction, and the receive diagnose function is used by the sink to accept (and transfer) the actual data.

The *send/receive protocol* implies a two-way (bidirectional) transmission of data from source to sink to source within a single transaction. The send/receive diagnose function initiates the source request (as with send), and the receive protocol is used by the sink to transfer the actual data. The transaction, in this case, is not terminated until the sink responds with a reply diagnose function. The reply causes data to be transferred from the sink machine's virtual storage to the source machine's virtual storage. The benefit of this protocol is that data may be exchanged within one transaction and result in less CPU execution time than two send protocol transactions.

The *sendx protocol* is provided to improve performance for one-way data transmission. This protocol eliminates the need for the sink machine to execute a receive function. The actual data is transferred into a special buffer at the same time the sink receives the external interrupt. A disadvantage of this protocol is that the sink virtual machine cannot be judicial with the message (data) it receives; i.e., the data is automatically "received" when the external interrupt is accepted.

The details of the various VMCF protocols are available in the *IBM Virtual Machine Facility/370 System Programmer's Guide*.[5] However, certain characteristics that should be noted are now described.

The asynchronous processing of a request is accomplished by the external interrupts. The source request causes an external interrupt to be queued for the sink (notification), and the completion of a transaction by the sink causes an external interrupt to be queued for the source (final response).

Because the entire process is asynchronous, the elapsed time between the send type and receive type functions could be significant (even indefinite). The source data and buffer, therefore, must be left intact until the entire operation has completed (is not buffered by the control program). The user parameter list, however, is copied into CP real storage and may be reused immediately following the diagnose instruction.

The processing within the source virtual machine may overlap the VMCF request. That is, the virtual machine is runnable (dispatchable) after the request is initiated. The source may send other messages to the same sink (requires unique message identifier) or send messages (data) to other virtual machines. Similarly, the sink virtual machine could be the source of other transactions in the system.

The amount of data transferred by a single request is restricted only by the size of the virtual machine (virtual storage size).

There may be multiple VMCF external interrupts queued for a single virtual machine (send and response). The priority of messages in the system is essentially determined by the order in which the external interrupts are received (may be FIFO (first in, first out) or by priority). Beyond that the sink virtual machine may select specific messages based on its own priority order (by user and message identifiers). That is, the sink may receive multiple external interrupts (requests) before ever responding to a single-specific request. The machine then can be selective in the messages it chooses and the order in which they are chosen.

VMCF is typically viewed as a mechanism for communicating and transferring data between distinct virtual machines. The interface does, however, support a logical wrap-around connection that allows a single virtual machine to send messages and data to itself (same as CTCA being connected to two channels on the same CPU). Since the wrap connection is supported, the interface does not preclude the possibility of connecting isolated components of a single virtual machine (e.g., operating system tasks).

VMCF data transfer functions may be executed with a zero data length. The interface may be used simply as a shoulder-tap or posting mechanism without regard for data, other than that provided by the message header.

control functions

VMCF includes several control functions (diagnose subfunctions) to assist in managing the interface and/or a specific application. Authorize and unauthorize functions are available to control access to or relinquish use of the interface, respectively. Quiesce and resume functions are provided to control message traffic at any given instant. Transactions may be prematurely terminated with a cancel function or rejected with the reject function. A fast path transmission of limited control data is possible with the identify function.

user doubleword and security

VMCF provides a special field within the user parameter list (doubleword) that may contain any desired user data. The doubleword is transmitted within the external interrupt message header. The significance of the doubleword is that it may be exchanged within

a single transaction, even with unidirectional protocol. For example, a source send request transmits the doubleword to the sink in the send external interrupt message header—a sink receive request will transmit a doubleword to the source in the response external interrupt message header.

The uses of such a feature can be diverse. The doubleword could be used to contain additional information for any request (e.g., a reason for reject). A user may define his own higher-level protocols, and the doubleword could be used for sequencing, pacing, and/or application control data. A most obvious use is that as a password or security code. The basic design of VMCF will allow any authorized user to access or send data to any other authorized user. The control of unwarranted access to a particular virtual machine is a function of the application itself. The specific authorization mechanism will allow a particular user to accept messages from *one* other user only (does not support multiple users). This protects a single user who is communicating with a specific machine but does not protect a machine that is communicating with many users (e.g., a central server). The use of the doubleword for this class of machine (as a password) could provide the necessary protection. A user could provide a password within his own data; however, this would require the data to be received before a validation could take place. An authorize specific function that allowed multiple user specifications (USERIDs) may satisfy some requirements but is somewhat limited in flexibility (all USERIDs of a particular application must be known).

data integrity

The integrity of a virtual machine is assisted by both the control program and the virtual machine software.[6] The integrity provided by the control program (in this case VMCF) is primarily a function of the design, structure, quality, and implementation of the code itself. The integrity of user data is largely controlled by the functions available to preserve and secure the data. A major consideration in the development of VMCF was to provide enough indicators to ensure that a user's data is protected from possible errors. The fact that the entire process is asynchronous, and can result in long delays between the time that a request is initiated and actually completed, opens a window to potential errors. For example, a user could initiate a request to another user and then be forced off the system (inadvertently or otherwise). Similarly, a paging I/O error could occur attempting to fetch a user page, or the user himself could violate the rules of the interface (e.g., storage protection, addressing, protocol). The important thing is that either the source or the sink is notified of the condition and in sufficient detail to allow for a recovery mechanism. VMCF provides a variety of return codes or error codes that detail possible conditions that can occur within the interface. The return codes are presented as a result of initiating a request via the diagnose instruction or as data transfer codes in the response external in-

terrupt message header. In the latter case, the code indicates that an error occurred after the successful initiation of a request via the diagnose instruction (e.g., sink user logged off or unauthorized without accepting or completing the request) and before the actual request had completed.

external interrupts

The VMCF interface is, for the most part, a software function. It does, however, involve the simulation or reflection of hardware external interrupts. Because this is true, the interrupts were made a part of the architecture in the same manner as other types of external interrupts:

- The interrupts are enabled through a combination of program status word Bit 7 (summary mask) and Control Register 0 Bit 31. Bit 31 is a special assignment for VMCF only; the other types of external interrupts are masked by their own bits in Control Register 0.
- The interrupts (and VMCF requests) will be purged or sent back to the source following any virtual system reset condition (e.g., a virtual initial program load).
- The interrupts are given a priority. In this case, VMCF interrupts are given the lowest possible priority as compared to other external interrupts in the architecture (e.g., interval timer, clock comparator, CPU timer, and external key).
- The interrupts were assigned an arbitrary code of X'4001'.

Performance

The performance characteristics of VMCF are a function of the internal path lengths, the virtual machine scheduler, and the application itself. In viewing the basic path lengths it became obvious that most of the code involved was not in VMCF, but rather in the linkage to get to VMCF—the reflection of external interrupts—and the scheduling and dispatching (task-switching) required to synchronize the execution of virtual machines. The path lengths could be reduced by changing some of the basic design criteria within the control program. The path required to get to the VMCF support module (DMKVMC)[7] requires (1) handling and interpreting a program check interrupt (privileged operation exception) occurring as the result of execution of the diagnose instruction, (2) passing control to a privileged instruction interpreter and simulator, (3) passing control to a diagnose instruction interpreter and preanalyzer, and (4) passing control to the VMCF module to execute the diagnose function. This path length, for example, could be significantly reduced by imposing few restrictions (e.g., not allowing a diagnose to be executed with the execute instruction) and giving control directly to the VMCF module from the program interrupt handler. Similarly, the virtual machine dispatcher is a control point within the control program where all functions

Table 1 VMCF performance characteristics (System/370, Model 158-3)

VMCF Protocol	*Transaction Rate* (seconds)	*Data Bytes Transferred*	*Source CPU Time* (milliseconds)	*Elapsed CPU Time* (milliseconds)	*VMCF Diagnose Subfunctions*	*Total Virtual CPU Time* (milliseconds)
	134.6	0		7.426	Send	0.995
Send	132.1	16,384		7.565	Send/Receive	1.015
	136.1	32,768		7.343	Sendx	0.999
					Identify	1.138
Maximum	136.6	4,096	2.646	7.319	Receive (128 bytes)	1.277
					Reply (128 bytes)	1.354
	112.5	0		8.884	Authorize	0.824
Send/Receive	114.4	16,384		8.736	Unauthorize	0.897
	112.8	32,768		8.863	Cancel	1.002
					Reject	0.947
Maximum	114.5	20,480	2.634	8.731	Quiesce	0.819
					Resume	0.826
	155.0	0		6.449		
Sendx	154.6	16,384		6.465		
	155.1	32,768		6.444		
Maximum	155.9	20,480	2.636	6.413		

eventually end. The dispatcher, among other things, provides the serialization and synchronization of internal processing and virtual machine execution (including the simulation of certain hardware interrupts). The path lengths involved to get to or from the dispatcher and required functions within the dispatcher could be eliminated by executing those functions directly in the VMCF module, i.e., interrupt reflection, task-switching, and redispatch. To do so, however, would violate important design elements of CP.

A virtual machine using VMCF may be, in essence, a logical extension of another virtual machine. The CP scheduler and dispatcher are unaware of VMCF and treat both machines as separate and distinct entities. The effects of scheduling functions on VMCF performance is determined largely by the overall system load (CPU and storage contention). The ostensible, higher-priority interactive user (who is simulating a terminal through the VMCF interface) may be treated as noninteractive (lower priority) since there is no real terminal activity associated with the virtual machine. Similarly, the control program is unaware of a VMCF simulated I/O operation, and a user may be dropped from queue even though there is logical active I/O. The adverse effect of this would be most apparent when the system is overloaded and a large number of users are contending for common resources.

performance characteristics

VMCF performance as related to the application itself is controlled by the size of the virtual machine and data (if paging), the protocols used, the frequency of VMCF requests, the structure of

code, and the implementation of the application. Table 1 provides the aggregate transaction (data) rates for each protocol and the CPU execution time for each diagnose function. The numbers were produced on a four-megabyte System/370, Model 158-3 running Release 5 of VM/370 with the VM System Extension Program Product (SEPP).[5] The VM/370 system included some local modifications but none that were related to VMCF or had a significant impact on VMCF path lengths. There were approximately 25 users logged-on in the VM/370 system, and there was no paging activity. The numbers supplied are the maximum achieved during 51 separate benchmarks. Each benchmark involved executing the requests 128 times and averaging the results. The test environment included two VMCF virtual machines (source/sink). The items included in Table 1 are defined as follows:

Protocol—the actual protocol used for the measurement.

Transaction Rate—the number of transactions per second. This number was computed by dividing one second (in microseconds) by the average elapsed CPU time for each transaction.

Data Bytes—the number of bytes transferred with each individual transaction. In the case of the send/receive protocol the number is actually one half of the total bytes transferred. The send/receive protocol includes a reply that is equal to the send data length.

Source CPU Time—the average total virtual CPU time (problem and supervisor) charged to the source for each request. This number includes the time to execute the function, enter a wait-state (LPSW), and receive the final response external interrupt. The number was produced with a CP diagnose function.

Elapsed CPU Time—the average elapsed CPU execution time for each request (initiation to final response interrupt). This number was produced using the real time-of-day clock and represents the real (wall clock) execution time for both source and sink virtual machines (including all CP task-switching overhead).

Subfunctions—the individual VMCF diagnose functions (data transfer and control).

Total Virtual CPU Time—the total elapsed CPU time to execute the diagnose instruction only. This number was produced using the real time-of-day clock.

In viewing the VMCF performance table, note the following items:

- The amount of data bytes transferred appears to have a negligible impact on the execution time and transaction rates when

there is no paging. This is true primarily because the VMCF data transfer path is small as compared to the other elements involved (e.g., task-switching, interrupt reflection).

- The path lengths (instructions executed) may be approximated or deduced by dividing the total execution time by the average instruction execution time for the machine.
- The numbers provided are conservative and are in reality biased against VMCF. Since the process is asynchronous and the execution of a virtual machine may overlap a request, the path lengths required for task-switching may not be a function of VMCF itself. For example, when a request is initiated, the call to the dispatcher to post or reflect the external interrupt for the sink is necessary only when the sink is enabled for the interrupt. Otherwise, the process is a normal function of the sink machine being interrupted (for whatever reason). These numbers were produced by having the sink machine always wait for a request and the source machine wait for a response following a request. A virtual machine that is waiting for external interrupts only is also dropped from the dispatcher queue (additional code) which may not be necessary in a live situation (for example, the machine may have active I/O operations that will keep it in queue). The source machine also could continue to run following a request, and the overhead of entering a wait state would not be included in the VMCF path lengths. The VMCF queuing facilities were not utilized in this test environment. A reduction in queue drop and task-switching overhead could be realized when multiple transactions are stacked and processed at the same time.
- The transaction rate for the sendx protocol would appear to be about a 15 percent improvement over the send protocol. The difference in the two protocols is the receive operation which is not required for sendx. The total virtual CPU time for send and receive is reasonably close. It could be assumed, then, that send and receive combined account for approximately 30 to 40 percent of the total execution time, and the other elements, such as entering wait, reflecting interrupts, and task switch, account for 60 to 70 percent.
- It would appear that the send/receive protocol provides a 60 to 70 percent improvement over send for bidirectional data transfer (two send operations would be required to exchange data). The actual transaction rate may be doubled for the send/receive protocol if the comparison is made to equivalent send operations (a reply is included in this transaction).

performance synopsis

There are potentially a large number of factors that may contribute to VMCF performance. The numbers supplied here should be viewed as an approximation and will differ from system to system. It is clear that a reduction in VMCF path lengths would improve performance but not as significantly as improving the other elements involved (e.g., task-switching, interrupt reflec-

tion). A more practical approach would be to eliminate the other elements altogether. This could be achieved by enhanced scheduling algorithms (VMCF awareness) and/or the inclusion of a synchronous protocol or no-response protocol for short messages. A synchronous protocol would imply that a virtual machine would not receive control back from the diagnose instruction, such as send, until the entire operation was complete. This would eliminate the paths for entering a wait and receiving the response interrupt. A no-response protocol would imply that the control program would buffer the message, thus eliminating the requirement for a response interrupt. In this case, the source virtual machine could run immediately following the diagnose instruction and before the sink actually received the data. The use of such protocols would clearly have performance advantages yet could be much more difficult to manage and control (e.g., the synchronous process may require that the source user wait for long periods of time; the no-response process would not provide indicators for errors occurring after the request was initiated).

VMCF applications

The potential application(s) for VMCF are many if viewed by the specific implementation. The purpose of this discussion is to give an overview of potential system uses of VMCF with little discussion of the actual implementation. In a broadened sense, these uses may be categorized as follows: (1) virtual subsystems and/or extensions to VM/370, (2) resource sharing, (3) multitasking/multiprogramming, (4) testing, and (5) intravirtual machine communication.

There have been specific implementations within each of these categories, some that were developed before the advent of VMCF and that use functions similar to those provided by VMCF. The potential uses are not described in a priority order, although the first appears to be the most widely used.

virtual subsystems and/or extensions to VM/370

A virtual machine subsystem may be thought of as any component running in a virtual machine that provides system-wide services to other virtual machines or users. A subsystem may be an extension of other components within the system (e.g., the control program) by providing functions that could be a part of the other components themselves. Examples of well-known virtual machine subsystems are RSCS (remote spooling communications subsystem), VNET, and CMSBATCH.[8,9] RSCS and VNET provide RJE (remote job entry) and/or network-type services, whereas CMSBATCH provides services relating to the background execution of jobs (e.g., compilations). The important element, as related to VMCF, is that subsystems in general require some type of intermachine communication media to provide their basic ser-

vices. The above specific subsystems use the spool file system as the media. This is appropriate, in these cases, because the data files may be classified as critical and require a store-and-forward technique. The noncritical data (e.g., query status of a particular link or job) could be implemented through a VMCF-like facility even for these particular subsystems.

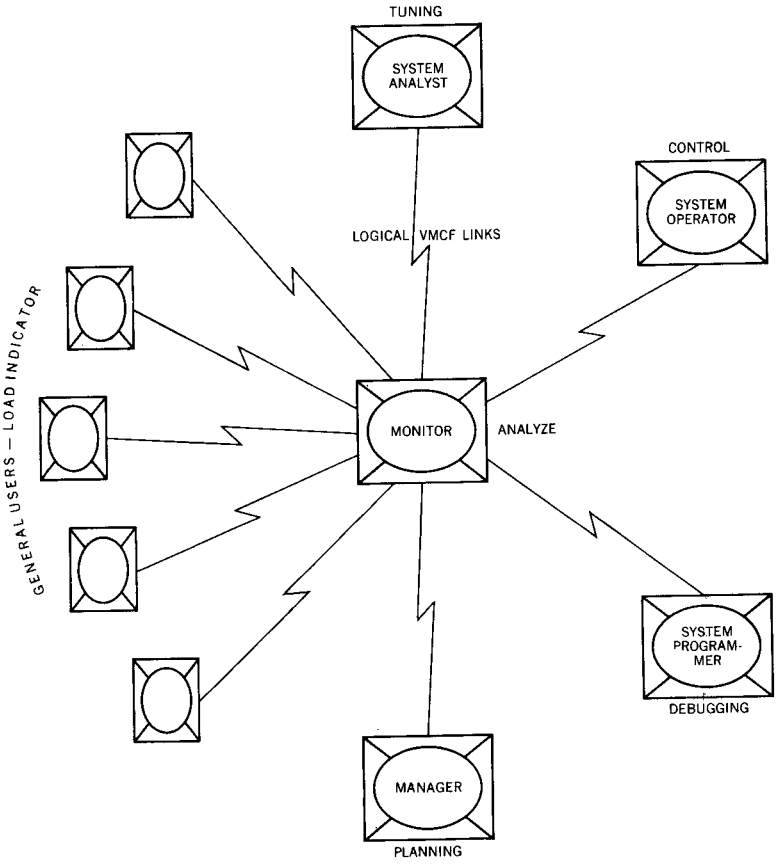
Subsystems have advantages in that they are logically isolated from other virtual machines yet easily accessible, simpler to maintain, and have little impact on the control program itself, i.e., CP. The disadvantage of subsystems may be performance. In some cases the performance impact can be minimized by features such as microcode assist which provides a significant reduction in the control program overhead required to support a virtual machine.

A hypothetical example of a subsystem could be an entire operating system, such as VS1, that runs in a virtual machine. CMS, the interactive component of VM/370, simulates some operating system functions (e.g., OS macros and associated SVCs, VSAM, DOS/VS) but does not simulate many other functions (e.g., data base and languages, write access to data sets).[9] It is obvious that an attempt to simulate all the available functions of an entire operating system would in fact be a duplication of the system itself. The original intent of CMS was to simulate operating system functions to a level of supporting OS compilers such as Assembler, PL/I, and COBOL. The functions provided by VMCF do not preclude a bridge between a CMS user and a virtual operating system such that the functions provided by the operating system could be made available to the CMS user as though they were simulated by the CMS machine. Similarly, operating system subsystems (e.g., VTAM) could be exploited in a like manner. However, the operating systems do not support VMCF in a virtual machine and would require the following modifications to do so:

- The external interrupt handler would have to be modified to recognize the VMCF external interrupt (Code X'4001') and post a VMCF control task (user program which has supervisor privileges).
- Modifications may be required to preserve the VMCF external interrupt mask (Bit 31 of Control Register 0). The operating system would be unaware of this bit since it is not part of the architecture of the hardware.

The application of a virtual machine subsystem could range from a simple command processor (e.g., virtual extensions to the CP command language) to a large shared data base system (e.g., IMS, CICS, and DL/I[10,11]). Figure 4 is an example of a recent prototype developed at the IBM San Jose Research Laboratory Computing Center. This prototype is used as an example because its use of

Figure 4 Multiaccess real-time monitor



VMCF is simple and the implementation did not require modifications to existing system control program software (CP or CMS). The prototype shown is a real-time performance monitor, debugging tool, and operations support tool. The application involves a large number of users. VMCF provides a multiaccess capability that allows many users to share a single program and data base. A simple CMS module was written to provide the terminal command interface and send commands to the monitor by means of VMCF. The monitor (which also accepts commands from a real terminal) required simple logic to handle and remember VMCF requests and respond through VMCF rather than display data on the real terminal. The CMS/VMCF interface module would do the actual displaying of data on the terminal associated with the user executing the request. This application reveals several qualities that could be applied to other types of subsystems:

- The monitor itself requires a special privileged class to extract internal data from the control program. The extracted data is

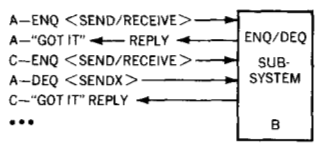
not privileged, yet to produce the data from any virtual machine would require a class that is not available to the general user. The VMCF interface enabled the general user to display data that otherwise would have required a privileged virtual machine.

- The monitor virtual machine need not be bound by a physical terminal, and simple logic was required to provide a multiaccess environment with a varied number of terminal types (those supported by CP). This same logic could be applied to any program(s) that runs in a virtual machine.
- The performance benefits of running a single copy of the program (the monitor) were significant as compared to running several copies of the program in separate virtual machines. The multiaccess capability provided the benefit by allowing the monitor to be run in a background virtual machine without disruption of the normal CMS. The value of the monitor was also significantly enhanced by making it available to a broader range of users.
- The ability to access the monitor through a CMS command (VMCF interface module) was a natural for automating and executing commands within a CMS EXEC procedure (automatic command execution facility). In this case, each user could create his own automatic monitor if desired (the monitor produces a variety of diffferent displays).
- Most of the CPU time required to access the monitor was charged to the user executing the request/command, and the monitor itself was not disrupted by real terminal I/O operations.

resource sharing

A resource, as viewed by a user, may range anywhere from a single data byte to a string of hardware devices. The control program does not provide a concise mechanism whereby users may serialize their own resources or data. The control program does have an internal locking mechanism that is used to serialize events within itself, but it is not available to the general user. Release 4 of VM/370 provided a capability to serialize access to a minidisk through a virtual (simulated) reserve/release feature. This feature is implemented through the virtual I/O interface (as on a real machine) and requires that an entire volume (or minidisk) be serialized rather than specific data (or data sets) within.

Figure 5 Hypothetical ENQ/DEQ subsystem



VMCF could conceivably be used to provide a logical enqueue/dequeue (ENQ/DEQ) facility such as that provided by OS-like operating systems. This may involve a distinct virtual machine (or subsystem) that manages the interface. The VMCF functions provide the communication media to control the process. A hypothetical example shown in Figure 5 could be as follows:

1. User A may execute a send/receive request to User B, the controller, with a bit defined in the user doubleword indicating

the type of request (e.g., ENQ-WAIT or ENQ-TEST). The actual data sent with the request would be the qualified name of the resource.

2. User B would then receive the data and determine if the resource was available (function of its own queues). If the resource were available, the controller could then reply to the enqueue request indicating that user A now has the resource.
3. User C could then execute a send/receive enqueue request for the same resource in use by User A. User B, or the controller in this case, would not execute a reply at this time since the resource is not available. User C could continue to run and overlap the request or enter a wait state for the response interrupt and reply.
4. User A could then execute a sendx request to the controller User B indicating a dequeue of the subject resource. User B may then respond to the User C request with a reply indicating that the resource is now available.

Further actions would continue in a similar manner.

The integrity of this type of facility is clearly the responsibility of its users. The serialization of resources involves an agreement among all concerned (which is true with any enqueue/dequeue facility). In this case, the actual controls are implemented by the user including such things as deadlock detection. The removal of such a procedure from the control program is advantageous since it does not involve real free storage demands controlled by an arbitrary limitation.

The serialization of a resource may simply be a built-in function of the subsystem controlling the resource. The monitor implementation discussed earlier serializes access to the program through the normal VMCF queues, i.e., it does not enable for external interrupts during processing of any given request. This is possible because there are not long delays between the time that a request is executed and a response is produced.

multitasking/ multiprogramming

A virtual machine is, in fact, a schedulable and dispatchable unit of work such as an operating system task. The execution characteristics of the machine may be controlled by a priority and guaranteed a certain amount of CPU time (when available). The elements involved in a multiprogramming system may include a serialization technique (ENQ/DEQ), a synchronization process (WAIT/POST), common storage, and a mechanism for dynamically attaching and detaching tasks. The ingredients for creating such a process between virtual machines is possible with VMCF. The control program provides an autolog command that may approximate the attach command. The autolog command will allow a virtual machine to be created (logged on) without a terminal and to the specifications of a predefined profile. Parameters may be supplied to

the virtual machine to control the program or programs that are executed. The force or logoff commands may be used to approximate the detach command, i.e., it terminates the virtual machine (see Figure 6). The VMCF functions may provide the connection of common storage and the synchronization and serialization processes. The control program provides the facilities to weight the execution of tasks, i.e., priorities, biases, and guarantees.

A simplified example of this process may be the CMS batch monitor.

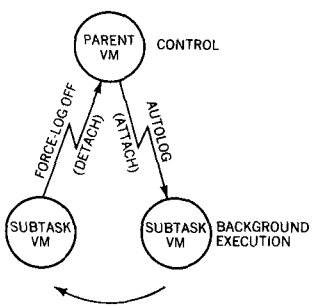
testing

In many cases the testing (particularly performance) of a component requires external controls that have little impact on the component itself. The small amount of code required to implement VMCF (and associated overhead) provides a nice "hook" into a system that can be controlled by an external interface or virtual machine. The shoulder-tap capability may be used for simple "wake-up" of a virtual machine. The process could also involve a comprehensive external simulator. For example, a SIO instruction could be replaced by a VMCF request, and a mechanism could be employed that simulates an entire I/O operation without actually requiring the hardware device. The fact that the process is external to the target allows the simulation process to be manipulated, controlled, and maintained in an isolated virtual machine.

intravirtual machine communication

This process involves connecting isolated components of a single virtual machine. As described earlier in this paper, VMCF supports a logical wrap-around connection that allows a virtual machine to communicate with itself (e.g., a CTCA connected to two channels on the same CPU). The uses of this facility may be similar to the CTCA environment. It is conceivable that a primitive multitasking system running in a virtual machine could use this facility to connect separated tasks (if such a feature is not available). The process may also have its advantages for testing such as that done between virtual machines.

Figure 6 Virtual machine subtask



Synopsis and conclusions

VMCF, then, is an interface that provides a mechanism to transfer data and communicate between virtual machines. The interface is a software process that is not bound by the external elements of a data processing system. VMCF is localized to a single real CPU running any number of virtual machines and is viewed by the user (programmer) as consisting of certain interface components:

- A diagnose instruction to invoke a specific function.
- A user parameter list to describe the function.
- External interrupts to synchronize the functions.

- A message header to describe external interrupts and data transmission characteristics.
- User data.

The characteristics of data transfer between virtual machines is a function of the protocol used for any given transaction. VMCF provides several protocols that allow for one-way or two-way data transmission. Control functions are available to assist a user in managing the interface. The mechanics of VMCF are efficient as an asynchronous process for communicating between virtual machines. The good points of its predecessor (virtual CTCA) were retained and improvements or extensions were made. Several features are provided to ensure a high degree of user data integrity and security.

The potential applications for VMCF were described at the system level and categorized as follows:

- Virtual machine subsystems (logical extensions to VM/370).
- Resource sharing (serialization).
- Multiprogramming (virtual machine subtasks).
- Testing (externally controlled simulation).
- Intravirtual machine communication (task to task).

VMCF is a base for future virtual machine applications, many that have yet to be conceived.

ACKNOWLEDGMENTS

The original prototype for VMCF was developed by A. N. Chandra at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. The Chandra prototype was later developed by the VM/370 Development Group.

I am indebted to H. M. Gladney and C. G. Colas, IBM Research, San Jose, for prompting me to write this paper, to G. Strickland, IBM Data Processing Division, Palo Alto, for giving me the opportunity to write the paper, and to the referees for their useful and constructive comments regarding this paper.

CITED REFERENCES AND NOTE

1. L. H. Seawright and R. A. MacKinnon, "VM/370—a study of multiplicity and usefulness," *IBM Systems Journal* **18,** No. 1, 4–17 (1979, this issue).
2. *IBM Virtual Machine Facility/370 Introduction*, GC20-1800, IBM Corporation, Data Processing Division, White Plains, NY 10604.
3. *IBM Virtual Machine Facility/370: Operating Systems in a Virtual Machine*, GC20-1821, IBM Corporation, Data Processing Division, White Plains, NY 10604.
4. *IBM System/370 Principles of Operation*, GA22-7000, IBM Corporation, Data Processing Division, White Plains, NY 10604.
5. *IBM Virtual Machine Facility/370: System Programmer's Guide*, GC20-1807, IBM Corporation, Data Processing Division, White Plains, NY 10604.

6. C. R. Attanasio, P. W. Markstein, and R. J. Phillips, "Penetrating an operating system: a study of VM/370 integrity," *IBM Systems Journal* **15,** No. 1, 102-116 (1976).
7. Source listing for module DMKVMC. This is the primary VMCF support module; it contains a SCRIPT prologue which details its functions.
8. E. C. Hendricks and T. C. Hartmann, "Evolution of a virtual machine subsystem," *IBM Systems Journal* **18,** No. 1, 111-142 (1979, this issue).
9. *IBM Virtual Machine Facility/370: CMS Users Guide*, GC20-1819, IBM Corporation, Data Processing Division, White Plains, NY 10604.
10. W. C. McGee, "The information management system IMS/VS," *IBM Systems Journal* **16,** No. 2, 84-168 (1977).
11. D. J. Eade, P. Homan, and J. H. Jones, "CICS/VS and its role in Systems Network Architecture," *IBM Systems Journal* **16,** No. 3, 258-286 (1977).

GENERAL REFERENCES

IBM Virtual Machine Facility/370: System Logic and Problem Determination Guide, SY20-0886, IBM Corporation, Data Processing Division, White Plains, NY 10604.

M. McGrath, "Virtual machine computing in an engineering environment," *IBM Systems Journal* **11,** No. 2, 131-149 (1972).

R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal* **11,** No. 2, 99-130 (1972).

Reprint Order No. G321-5087.