ESA/390 interpretive-execution architecture, foundation for VM/ESA

by D. L. Osisek
K. M. Jackson
P. H. Gum

The interpretive-execution facility of Enterprise Systems Architecture/390™ (ESA/390™) provides an instruction for the execution of virtual machines. This instruction, called START INTERPRETIVE EXECUTION (SIE), was initially created for virtualizing either System/370™ or 370-XA architectures, and was used later for virtualizing ESA/370™ and ESA/390 architectures. SIE has evolved to provide capabilities for a number of specialized performance environments. Most recently it provides for the unique requirements of Enterprise Systems Architecture/Extended Configuration (ESA/XC) virtual-machine architecture. This comprehensive set of capabilities in the architecture serves as the platform for the ability of VM/ESA™ to provide functions in virtual machines for end users and system servers. This paper describes the evolution of SIE and outlines use of the various capabilities in VM/ESA.

The Virtual Machine/Enterprise Systems Architecture™ (VM/ESA™) product uses the ESA/390™ interpretive-execution facility[1,2] to establish the virtual-machine environment. In this environment, the processor directly executes most of the functions of the virtual machine.

The evolution of the interpretive-execution architecture, including the special facilities for high-performance virtual machines provided with the Processor Resource/Systems Manager™ (PR/SM™) feature, is reviewed. The purpose of each development is discussed, and the VM Data Spaces architecture is described. VM Data Spaces is the architecture underlying the recently announced Enterprise Systems Architecture/Extended Configuration (ESA/XC) virtual-machine architecture.[3] Finally, this paper describes the procedures used by the control program (CP) portion of VM/ESA[4] to manage interpretive execution and control virtual-machine functions that are not provided by the real machine, including the support of the ESA/XC architecture.

Interpretive-execution architecture

Within this paper, the term *guest* refers to any virtual, or "interpreted" machine.[5] The control program directly managing the real machine is referred to as the *host* and is responsible for establishing the guest execution environment. The machine is placed in the interpretive-execution mode by the host, which issues a single instruction, START INTERPRETIVE EXECUTION (SIE). In this mode, the machine provides the functions of a selected architecture. This architecture may also be available on a real machine, such as System/370™, 370-Extended Architecture (370-XA), Enterprise Systems Architecture/370™ (ESA/370™), or Enterprise Systems Architecture/390™ (ESA/390™). Alternatively, the architecture may be provided exclusively in the virtual-machine environment, such as the ESA/XC architecture. The functions provided include execution of privileged and problem-program instructions, address translation, interruption handling, and timing among other things, and I/O in some cases. The machine is said to *interpret* the functions that it executes in the context of the virtual machine. Special-purpose hardware allows interpretation to proceed at speeds comparable to "native" execution. (*Native* denotes the architecture outside the interpretive environment.) Similarly, many types of interruptions are interpreted—presented directly to the guest by the machine—without host intervention.

Simulation attempts to "execute" guest functions transparently.

In the virtual-machine environment, the guest program perceives the full complement of functions defined for the designated architecture. Most of the functions are provided in the form of the interpretive-execution facility. The remaining functions are provided by the underlying host control program, called CP for VM/ESA, through a process called *simulation*. Except for the processing time required, simulation attempts to "execute" guest functions transparently, so that it is indistinguishable to the guest program whether a function is performed by the machine or the host.

The operand of the SIE instruction, called the *state description*, contains information relevant to the current state of the guest. When execution of SIE ends, information representing the state of the guest, including the guest program status word (PSW), is saved in the state description before control is returned to the host. This information is used and modified by the host during simulation and is used later to resume execution of the guest. Other information in the state description determines the mode and other environmental conditions in which the guest is to execute.

The ESA/390 interpretive-execution architecture can place the machine in one of three architecture interpretation modes: System/370, ESA/390, or VM Data Spaces mode. However, these three machine modes supply the functions needed for offering five modes to guests. In addition to System/370 and ESA/390, 370-XA and ESA/370 are naturally provided, each as a subset of ESA/390. ESA/XC arises from the cooperation between the machine in VM Data Spaces mode and software support in VM/ESA CP.

While in interpretive-execution mode, a virtual machine is constrained to a portion of the real-machine resources, as allocated by the host.

- Guest storage is confined either to some portion of host real storage or to host virtual address spaces controlled by the host system.
- Host enabled and disabled states are generally undisturbed by execution of the guest.
- Host timing facilities are also undisturbed; instead a second set is provided for the guest.
- One complete and logically separate set of control registers is maintained by the machine for use by the host and another is maintained for use by the guest. Other registers are shared between the host and guest.

This protection of the host from interference by the guest permits the host to meet its primary responsibility of efficiently parceling out the real resources to multiple guests, and prevents one guest from interfering with another.

When first introduced with 370-XA, SIE provided two similar but distinct architectures for the virtual machine. One was the System/370 architecture. Because of earlier successes with several assists for Virtual Machine/370 (VM/370), full machine interpretation of nearly all the privileged operations was provided, with the notable exception of the I/O instructions. The second interpreted architecture was the new 370-XA architecture itself. Offering both architectures side-by-side in virtual machines provided a migration path from the earlier architecture as well as a test environment for the new architecture. As was the case with the native architecture, interpretive-execution architecture subsequently evolved to incorporate ESA/390 as a replacement for 370-XA and ESA/370.

Representing guest absolute storage. Fundamental to any architecture is the method for providing access to storage. The method for representing absolute[6] storage is a key consideration for virtual machines. Two basic storage modes are provided by the interpretive-execution architecture: preferred-storage mode and pageable-storage mode. In preferred-storage mode, a contiguous block of host absolute storage is assigned to the guest, whereas in pageable-storage mode, dynamic address translation (DAT)[7] at the host level is used to map guest main storage.

Preferred-storage mode. Existing batch systems maintain relatively high I/O rates and are efficient managers of real storage. For these guests, preferred-storage mode is a good means to provide production levels of operation. In this mode, the lower addresses of the machine storage are dedicated to the guest. However, this scheme limits the number of guests operating with this level of performance to one.

The I/O for these guests, or at least their channel programs, is handled directly by the machine. The interpretive-execution environment assures that the host program is immune from errant guest operations, including errant I/O operations. This is an important characteristic: preferred-storage mode assures the integrity of the overall system while at the same time allowing the guest to operate with a subset of the real-machine resources at near-native performance levels.

With early releases of VM, the special preferred-storage-mode guest paid a performance penalty for the privilege of executing as a virtual machine. This was due largely to the host-control-program service of handling I/O instructions and returning the interruptions. For VM/370, the Preferred Machine Assist (PMA) stepped into this breach. With VM/XA, SIE Assist was introduced to provide machine execution of guest I/O instructions and route I/O interruptions directly to the guest. SIE Assist performed the I/O for both System/370-mode and 370-XA-mode guests. SIE Assist also provided the authorization checks needed to preserve integrity for the host system and other guests.

Pageable-storage mode. Pageable-storage mode is the second method provided by the interpretive-execution architecture for representing guest absolute storage. The host has the ability to scatter the real storage of pageable-storage-mode guests to usable frames anywhere in host real storage by using the host DAT, and to page guest data out to auxiliary storage. This method provides flexibility when allocating real-machine resources while preserving the expected appearance of a contiguous range of absolute storage for the guest.

Guest dynamic address translation. A virtual-machine environment may call for application of DAT twice: first at the guest level, to translate a guest virtual address through guest-managed translation tables into a guest real address, and then, for a pageable guest, at the host level, to translate the corresponding host virtual address to a host real address.

In VM/370, the need to effect two levels of address translation for pageable virtual machines with guest DAT active was satisfied by means of *shadow translation tables*, segment and page tables built by the host reflecting the combined results of the two mappings. The increased addressing capacity offered by 370-XA threatened to limit the performance achievable through shadow mechanisms, because of the possible sparseness of address references over the much larger two-gigabyte address range, and because of the larger translation-table sizes. Another consideration was the cost to maintain and ensure the integrity of the shadow tables.

These concerns led IBM to forsake shadow tables for general use in interpretive execution, in favor of performing both levels of translation in the machine. As with the native architecture, translation-lookaside buffers are built into the machine to retain the results of previous address translations, and so speed the resolution of addresses in pages that are repeatedly referenced.

Controlling guest execution. In certain cases, the host must intercede in operations normally delegated to the machine. For this purpose, the state description includes controls settable by the host to "trap," or *intercept*, specific conditions. *Interception control* bits request that the machine return control to host simulation when particular guest instructions are encountered. *Intervention controls* capture the introduction of an enabled state into the PSW, so that the host can present an interruption which it holds pending for the guest. Intervention controls may be set asynchronously by the host on another real processor while interpretation proceeds. The machine periodically refetches the controls from storage, so that updated values will be recognized. Guest interruptions can thereby be made pending without prematurely disturbing interpretation.

Guest multiprocessing. In contrast with earlier virtual-machine support, the interpretive-execution architecture supported guest multiprocessing from the beginning. As a consequence, a virtual machine that employs multiprocessing receives a substantial boost in capacity. Prefixing, or the ability to assign the first 4K range of addresses to a distinct 4K-byte block of absolute storage for each virtual CPU, is an integral part of multiprocessing. Thus, with interpretive execution, prefixing is standard.

Simulating guest instructions in a multiprocessing virtual-machine environment requires special consideration. Simulation is provided by the host program following an interception from one of the virtual CPUs. This simulation may require resolving a guest virtual address. This is typical when the guest is Multiple Virtual Storage (MVS). However, if the result of the translation is held even briefly in a host register or table, that register or table constitutes a (virtual) translation-lookaside buffer. What if another virtual CPU issues an INVALIDATE PAGE TABLE ENTRY (IPTE) instruction during this brief period, in preparation for reassigning the guest real page frame? The IPTE might invalidate the translation held by the host on behalf of the first guest CPU. If the IPTE were allowed to finish and the guest program to continue executing, the guest would erroneously conclude that all references to the target guest virtual address using the old page-table entry (PTE) contents were complete. An interlock bit is provided for dealing with this situation. IPTE is required to test and set the interlock bit by interlocked means. The host agrees to set this interlock before simulating guest DAT (in the course of simulating a guest instruction) and to leave it set until it has finished using the results of that simulation (the "lookaside") to access guest storage. If the page-table entry is locked via the interlock bit when an IPTE instruction is interpreted, the IPTE instruction is intercepted to permit host program resolution.

To maximize parallelism in a virtual multiprocessor, VM/ESA manages the word containing the IPTE interlock bit as a shared/exclusive lockword. Each simulation which performs the guest DAT obtains a share of the lock; IPTE interpretation and simulation obtain an exclusive hold. Thus multiple DAT simulations can occur simultaneously, and only a guest IPTE causes serialization.

Multiple high-performance guests. With achievable performance for the preferred-storage-mode guest near native performance, as described earlier, attention turned to providing the same for more than one guest. Multiple Domain Facility™

The interpretive-execution architecture supported guest multiprocessing from the beginning.

(MDF™) was the first product to achieve concurrent execution of two or more operating systems with high performance on a single shared central computing complex. However, IBM was also working in the same direction, leading to the Processor Resource/Systems Manager (PR/SM) feature.[8] PR/SM permits either flexible hardware partitioning or multiple high-performance guests under VM. The value of this development, of course, lies in the ability to reassign real resources dynamically with minimal performance penalty, for use under a variety of different architectures or systems. Initially up to four high-performance guests were supported, each running in a separate *zone*. This was subsequently extended to six zones.

Zone relocation. The most important consideration in extending support from one preferred-storage-mode guest to multiple guests is the means for translating addresses for I/O purposes. With high-performance I/O devices, little time is available to perform the translation; in many cases page faults would result in loss of data. In these circumstances a single-register-translation mechanism serves very well.

Single-register relocation, or zone relocation, uses two values to translate a guest absolute address to a host absolute address. One value is an upper limit, the maximum value in the zone. The second value is a lower limit which is added to the guest absolute address to produce a host absolute address. The two values thus define a zone between a beginning and ending address within the absolute storage available on the machine.

Region relocation and interpreted SIE. In the original offering which permitted multiple high-performance guests, the channel used zone relocation while the CPU continued to use DAT to map contiguously the same storage specified for the zone. With this arrangement, all guest systems except for the case when VM/ESA (or VM/XA) is a guest of itself could be provided at acceptable production levels of performance.

Steps have been taken in the evolution of the architecture to support CMS as well.

When VM/ESA is a guest of itself, there are three levels of programs: VM/ESA as the host, VM/ESA as the guest (called a first-level guest), and guests of the VM/ESA first-level guest. These second-level guests might themselves operate with the DAT. In this situation, address translation involves repetitive application of DAT, potentially requiring 27 storage references to translate an address into the corresponding absolute machine location and access the data. In addition, execution of second-level guests requires handling two levels of interpretive execution; software simulation of the second level of interpretive execution brings additional overhead. The costs of repeated translation and nested interpretive execution are clearly limiting factors in the performance achievable for VM/ESA as a guest.

Two new machine functions, Region Relocation and Interpreted SIE, were introduced to address these problems. Region Relocation replaces the lowest-level application of the DAT with zone relocation, as is used in the channel; this reduces the 27 storage references for DAT to the original nine needed for a first-level guest, plus nine additions. Interpreted SIE allows the machine to institute another instance of interpretive execution when the machine is already in interpretive-execution mode; that is, the guest SIE instruction and the second-level guest execution it requests can be interpreted by the machine, rather than simulated by the host.

These capabilities together provide performance levels for second-level guests that are comparable to the levels provided for first-level guests. As a result, one release of VM can run as a guest of another VM system at acceptable performance while migration to the new release occurs in stages.

Performance refinements for CMS guests. While there has been heavy focus on high-performance multiserver and batch guests, the Conversational Monitor System (CMS) has not been neglected. CMS is the personal-choice system to a large population of interactive users, and steps have been taken in the evolution of the architecture to support CMS as well.

Expedited SIE subset. VM/ESA uses the SIE instruction to specify a virtual machine for each CMS user. Similar to the high-performance cases, there are characteristics or uses of CMS that may be classified as typical. By optimizing the real-machine design for the most frequently encountered characteristics of a class of virtual machines, machine performance can be improved. Expedited SIE subset provides such optimizations for CMS guests. This development capitalizes on the fact that certain aspects of the architecture are used infrequently by CMS. For example, CMS never turns on DAT. The overhead for initializing these infrequently-used functions is therefore bypassed. Instead, attempts to use uninitialized functions are detected, and the cost to initialize any of these functions is incurred only when necessary.

SIE storage-key facility. The SIE storage-key facility is an enhancement to interpretive execution which defines an architected location in host tables for the key values of nonresident storage. This allows the machine to interpret guest instructions like SET STORAGE KEY for nonresident pages, rather than passing control to host simulation.

VM Data Spaces

VM Data Spaces is an extension to the interpretive-execution architecture that allows a pageable guest that does not otherwise use DAT to access data in multiple host address spaces. The alternate spaces could be "data spaces" created by this or other virtual machines, or primary spaces of other virtual machines. This memory-sharing mechanism is more efficient than message-passing protocols for communicating among virtual machines.

Access registers in ESA/370 and ESA/390. The ESA/370 architecture introduced access registers. These registers allow a problem-state program to refer to data in multiple address spaces concurrently, without supervisor intervention.[9,10] The access registers offer an improved method to move data between two address spaces. They also allow the use of the complete instruction set to operate on data in multiple address spaces.

In ESA/370 and ESA/390, as in System/370 and 370-XA, the base (B) field or register (R) field of an instruction designates a general register. In the access-register mode of ESA/370 and ESA/390, the same-numbered access register is used during access-register translation (ART) to determine the address space of the operand.

Access-register translation uses an access-list-entry token (ALET) in an access register to derive the segment-table designation (STD) to be used during dynamic address translation (DAT). The STD corresponds to an address space.

Access registers are also available to a guest in ESA/390 (or ESA/370) mode. The host is responsible for loading the guest's access-register values before starting interpretive execution, and for saving them (and restoring host values) afterward. The guest operating system must build the guest virtual address spaces and associated control structures, just as it would natively. Pages in these address spaces may be mapped to areas of guest main storage or paged by the guest supervisor to auxiliary storage.

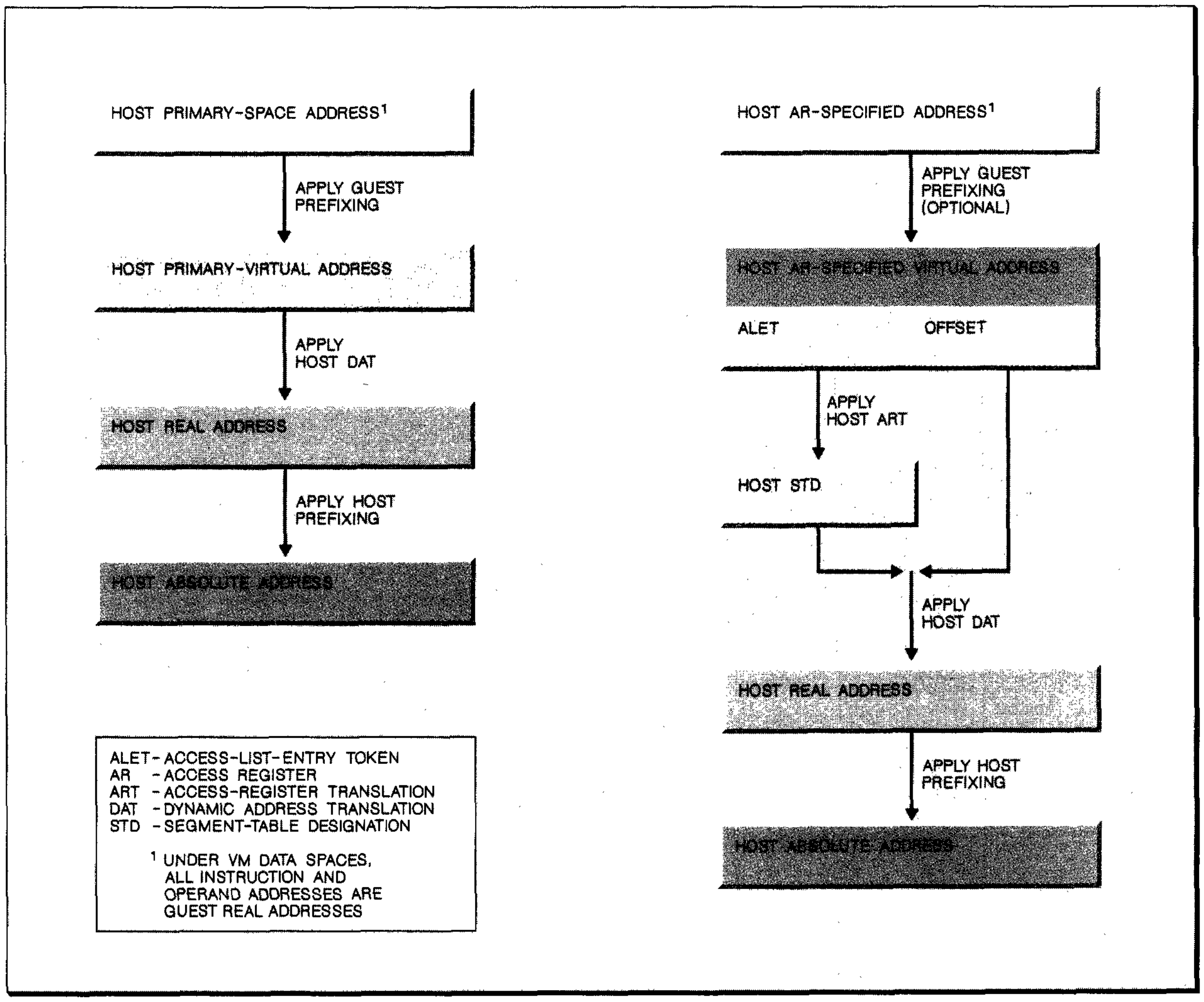
Motivation for VM Data Spaces. On an operating system like Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA™), the access registers introduced with ESA/370 bring a powerful new capability: the addressing of data in multiple address spaces in a sequence of instructions, or even in the same instruction, without control-program intervention. MVS/ESA runs programs with DAT enabled, so that the virtual addresses each program references are translated through MVS-maintained constructs into real addresses. Translation exceptions interrupt to MVS, which then pages in the required data. In this environment, access registers offer several benefits:

- Programs can directly manipulate much larger amounts of data.
- Programs have unrestricted access beyond the two-gigabyte primary address space. The full instruction set can be used to operate on data in any of the multiple spaces.
- Individual programs can share data, as authorized by the owning program. This is enforced by the operating system.
- Programs can segregate data more logically, keeping similar or related data in the same space, to facilitate controlled sharing.

The capabilities which access registers offer are attractive. However, the structure of VM is substantially different from that of MVS. VM has always been a "two-tiered" system: The control program (CP) component of VM creates a separate virtual machine for each logged-on user. Application programs run under the Conversational Monitor System (CMS), a single-user "second-level" operating system running within the user's virtual machine. A CMS virtual machine can support an interactive user, a system server like a file-system manager or network spooler, or a private server such as an advanced program-to-program communication (APPC) peer.

CP manages system resources, establishes the virtual-machine environment, and enforces isolation among the simulated machines. CMS assumes the responsibility for application services such as file and program management, and for interacting with the end user. CP applies authorization controls to bound the user's (virtual machine's) activities; CP uses architectural facilities like DAT and guest extent checking, as described above, to keep these boundaries secure. Conversely, CMS enforces few controls over the application program: Programs under CMS run in (virtual) supervisor state. CMS makes some use of storage keys to prevent inadvertent damage, but a "willful" program can always circumvent CMS and assume control of the virtual machine. According to traditional VM philosophy, each user's machine is the user's own. CP ensures that the acts of an errant or malicious program are confined to the virtual machine in which the program is run.

Figure 1 Operand-address-translation processes for host primary-space addresses vs host AR-specified addresses


HOST PRIMARY-SPACE ADDRESS[1]
APPLY GUEST PREFIXING
HOST PRIMARY-VIRTUAL ADDRESS
APPLY HOST DAT
HOST REAL ADDRESS
APPLY HOST PREFIXING
HOST ABSOLUTE ADDRESS
HOST AR-SPECIFIED ADDRESS[1]
APPLY GUEST PREFIXING (OPTIONAL)
HOST AR-SPECIFIED VIRTUAL ADDRESS
ALET
OFFSET
APPLY HOST ART
HOST STD
APPLY HOST DAT
HOST REAL ADDRESS
APPLY HOST PREFIXING
HOST ABSOLUTE ADDRESS
ALET - ACCESS-LIST-ENTRY TOKEN
AR - ACCESS REGISTER
ART - ACCESS-REGISTER TRANSLATION
DAT - DYNAMIC ADDRESS TRANSLATION
STD - SEGMENT-TABLE DESIGNATION
[1] UNDER VM DATA SPACES, ALL INSTRUCTION AND OPERAND ADDRESSES ARE GUEST REAL ADDRESSES

Moreover, CMS manages only the linear storage of a single virtual machine; that is, CMS runs without enabling DAT and does not manage (page or swap) virtual storage. These tasks are the domain of CP.

In short, the border between CP and CMS is the boundary for both authority and virtual-storage management. The VM Data Spaces facility is tailored to VM's unique two-tiered structure. It allows CP, the arbiter of authorization, to build the ART constructs which permit individual virtual machines to access multiple host virtual (i.e., guest absolute) address spaces. CP can pass back to the virtual machine an ALET designating the access to a space, and the program in the virtual machine (CMS or an application program) can then use that ALET to address an alternate guest absolute space. Since the ALETs used by the guest are translated through the ART constructs of CP, counterfeit ALETs have no adverse effect beyond the authorized scope of the individual guest. In ESA/390 real-machine architecture, ART is performed first to identify the address space, and then DAT is performed on that space's tables to find the real address of the data; likewise, in the VM Data Spaces architecture, host ART identifies the host virtual (guest absolute) space, and host DAT then derives the host real address of the target.

In contrast to the use of access registers by an ordinary ESA/390 guest (e.g., MVS/ESA), this scheme extends the scope of access to all address spaces which CP manages, rather than just those built by the virtual-machine supervisor. It thus allows controlled memory sharing across virtual-machine boundaries. The burden of authorization rests within CP, which cannot be circumvented. Finally, VM Data Spaces removes the limit of two gigabytes of guest absolute (host-managed) storage accessible to CMS.

PSW modes under VM Data Spaces. Under VM Data Spaces, a guest may be in either primary-space mode or access-register mode. Each mode determines how guest operand addresses are translated. In primary-space mode, guest operand addresses are resolved in the host primary address space. In access-register mode, guest operand addresses are resolved in any of up to 16 different address spaces concurrently, according to the values which the guest manages in the access registers. In either mode, instructions are fetched from the host primary address space. This is consistent with the handling of instruction addresses in the other modes of interpretive execution.

Address translation in primary-space mode is identical to address translation for a pageable-mode guest in ESA/390 mode with guest DAT off. The contents of access registers are ignored, and host access-register translation (host ART) is not applied.

When a guest is in access-register mode, operand addresses are called host access-register-specified (host AR-specified) addresses. A *host AR-specified address* consists of an ALET in an access register and an offset. The ALET designates an address space, and the offset selects a location within that space. Guest access-register translation and guest dynamic address translation are not used. Instead, host AR-specified addresses are translated through host access-register translation and host dynamic address translation to produce a host real address. Figure 1 contrasts the address translation processes for host primary-space addresses and host AR-specified addresses.

The contents of an access register designate the host virtual address space. If the access register contains zero, only guest prefixing, host DAT, and host prefixing are applied; the address is in the primary space.

Host access-register translation. When an access register contains a value other than zero and the guest is in access-register mode, the operand address specified refers to data in a host AR-specified address space. The contents of the base register together with the displacement and the index register, if applicable, are used to determine the offset of the data within the address space.

To resolve the address space of the operand, host access-register translation (host ART) is applied. Host access-register translation is similar to the access-register-translation process used in ESA/390 mode. Host ART uses an access-list-entry token in an access register to obtain the segment-table designation (STD) to be used during host dynamic address translation. Figure 2 shows a flow chart of the host ART process.

During host ART, the designated access register contains an access-list-entry token. This token, which is obtained using VM/ESA services, has an access-list-entry number.

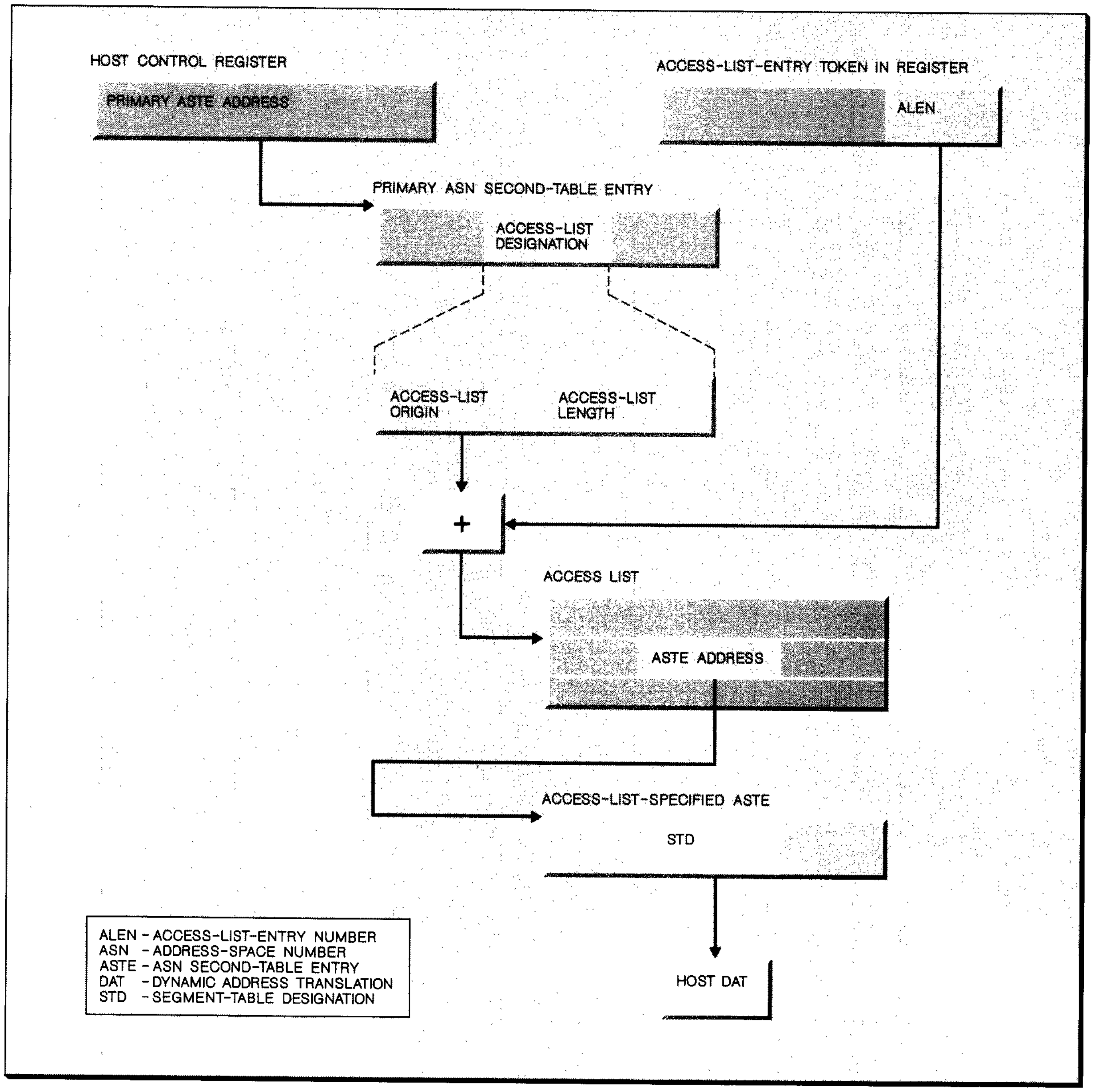
The origin of the primary address-space number (ASN) second-table entry (primary ASTE)[11] is obtained from a host control register. An ASTE is associated with each address space and has the same format in VM Data Spaces as in ESA/390. The primary ASTE contains the origin of the access list used during host access-register translation (host ART). An ASTE is also used later in the host ART process. This use will be discussed later in this section.

An access list contains entries which represent the addressing capabilities of the guest. The access-list-entry number in the access register together with the access-list origin in the primary ASTE determine the access-list entry to be used during host ART.

An access-list entry contains the address of an ASN second-table entry (ASTE). When an ASTE is located by an access-list entry, it is referred to as an access-list-specified ASTE, to distinguish this use of the ASTE from the primary ASTE described earlier. The access-list-specified ASTE contains the STD to be used during host DAT.

Private-space facility. ESA/370 introduced the concept of a "private space," a space that holds

Figure 2 Host access-register translation (host ART)


HOST CONTROL REGISTER
PRIMARY ASTE ADDRESS
ACCESS-LIST-ENTRY TOKEN IN REGISTER
ALEN
PRIMARY ASN SECOND-TABLE ENTRY
ACCESS-LIST DESIGNATION
ACCESS-LIST ORIGIN
ACCESS-LIST LENGTH
+
ACCESS LIST
ASTE ADDRESS
ACCESS-LIST-SPECIFIED ASTE
STD
HOST DAT
ALEN - ACCESS-LIST-ENTRY NUMBER
ASN - ADDRESS-SPACE NUMBER
ASTE - ASN SECOND-TABLE ENTRY
DAT - DYNAMIC ADDRESS TRANSLATION
STD - SEGMENT-TABLE DESIGNATION

no contents in common with other spaces. This was significant to MVS/ESA, which had previously imbedded the prefix area and portions of the supervisor into every address space it created, and had marked those segments "common" to improve lookaside efficiency. The ESA/370 private-space facility allowed MVS/ESA to construct data spaces devoid of these common segments, and to omit the special protection mechanisms for the low address range, which are needed for the prefix area. The private-space facility allows equal access to all addresses in a data space, without interference from low-address protection and fetch-protection override.

This private-space function is also useful to VM/ESA for the same reasons; indeed, VM Data Spaces extends its scope. In a virtual-machine-created data space, since the virtual machine's prefix area exists only in its host primary space, it is inappropriate to apply prefix-related protection mechanisms to that data space. Moreover, it is inappropriate to apply prefixing at all. If a data space is shared among virtual CPUs in a virtual multiprocessing configuration or among separate user virtual machines, each sharer may have a different prefix value. However, all sharing users should be able to use the same address to reach the same location in the data space. Therefore, VM Data Spaces extends the effect of the private-space attribute to suppress prefixing as well. Similarly, the size of the virtual machine's primary space that is specified by the main-storage extent in the state description has no bearing on the size of other spaces it may access. Thus, the private-space control in VM Data Spaces suppresses the application of main-storage-extent checking. In short, the private-space attribute gives each sharer of a data space the same "view" of the data.

The inherited name "private space" is somewhat a misnomer for VM/ESA. VM does not use architected common segments, and so does not need that effect of the "private" attribute. Moreover, in VM/ESA, "private" is not an attribute of the address space, but rather of the access. When a user's primary address space is shared with a second virtual machine, the owner's access is not marked private, since prefix effects are appropriate for the primary space. The sharer's access would be marked private, to suppress application of the sharer's prefix to the addresses in the foreign space.

Access-list-controlled protection. ESA/390 introduced a new function into ART: the ability to grant read-only access to an address space. With VM Data Spaces, this allows some guests to have read/write access to a space, and others read-only access. This represents a significant enhancement over the previous shared-memory capability in VM. Earlier releases of VM allowed "saved segments" to be imbedded into the absolute address spaces of multiple virtual machines, but the shared data were writeable either by all sharing users or by none of them. Control over an individual virtual machine's access allows a service machine, such as that for the CMS Shared File System, to load data securely into an address space, from which user machines can fetch the data directly.

Access-list-controlled protection is included in native ESA/390 as well as VM Data Spaces. Therefore, CP can use this mechanism, rather than explicit testing in software, to enforce protection when simulating an operation for a guest.

The private space function is also useful to VM/ESA.

Interestingly, the ability to grant read-only authority to an address space was not provided in ESA/370 and is not exploited by MVS/ESA. MVS/ESA can use storage keys to effect read-only access for selected users. However, VM gives over the assignment of storage keys and PSW keys to the guest in each virtual machine, making keys inadequate to enforce authority across virtual machines. Access-list-controlled protection addresses the unique needs of VM in this area.

Instruction execution in VM Data Spaces mode. The operation of certain DAT-related instructions is modified to permit their operation without DAT when the guest is in the VM Data Spaces mode. This allows an application program to use the same problem-program instructions to control host ART under VM Data Spaces as it uses to control native ART under ESA/390.

The INSERT ADDRESS SPACE CONTROL and SET ADDRESS SPACE CONTROL instructions are changed to be usable without DAT. INSERT ADDRESS SPACE CONTROL is used to obtain the current mode of the guest, either primary-space or access-register mode. SET ADDRESS SPACE CONTROL is used to set either primary-space or access-register mode.

The ESA/XC virtual-machine architecture. VM Data Spaces is an architectural mode unique to the virtual-machine environment. To take advantage of the facilities provided by VM Data Spaces requires software support in a host control program, such as VM/ESA CP. The specific virtual-machine interface arising from the collaboration between VM/ESA CP and the machine is termed the *Enterprise Systems Architecture/Extended Configuration* (ESA/XC) architecture. This name emphasizes the advance from traditional machine configurations, containing a single span of absolute storage, to an environment with multiple discrete absolute address ranges.

The thrust of the VM Data Spaces architecture is to keep knowledge of and responsibility for the control structures "below the line," in the host. The manifestation of access-register function to the guest should be more akin to an application-program interface than to that in the ESA/390 real-machine architecture. VM therefore defines a second new architecture, ESA/XC, as the interface to its virtual machines which exploit VM Data Spaces. The VM Data Spaces architecture precisely defines the interface between host and real machine; the ESA/XC architecture specifies the somewhat different interface between CP and the guest program (e.g., CMS or an application program).

Some examples may serve to clarify.

- VM Data Spaces defines the structure of an ALET and the meaning of each field in it, and the way it participates in the ART process. In contrast, an ESA/XC ALET is purely a token, a 32-bit value constructed by the host and presented to the guest. This abstraction allows future enhancements to change the format of the ALET with no effect on the ESA/XC architecture or the programs that use it.
- Just as native ART translates an ALET into a segment-table designation (STD), host ART of VM Data Spaces translates an ALET into a host STD, which is then used during host DAT. Conversely, host ART in ESA/XC is defined to map an ALET into an address-space-identification token (ASIT), which is the unique representation of a host space in ESA/XC. The ESA/XC guest has no knowledge of STDs or translation tables; it deals only with the ASIT as the unique identifier for the space and the ALET as a "handle" for access to the space.
- The host ART process may encounter various specific exception conditions, which are reported to the host to drive the proper handling. Conversely, the exception conditions reported to the ESA/XC guest are fewer and more generic, representing the problem in terms meaningful to the ESA/XC application.

Throughout the definition of ESA/XC, the interface is specified in the simple, clear form suitable for an access-register application, and nuances of the VM Data Spaces architecture which are not appropriate to that environment are concealed.

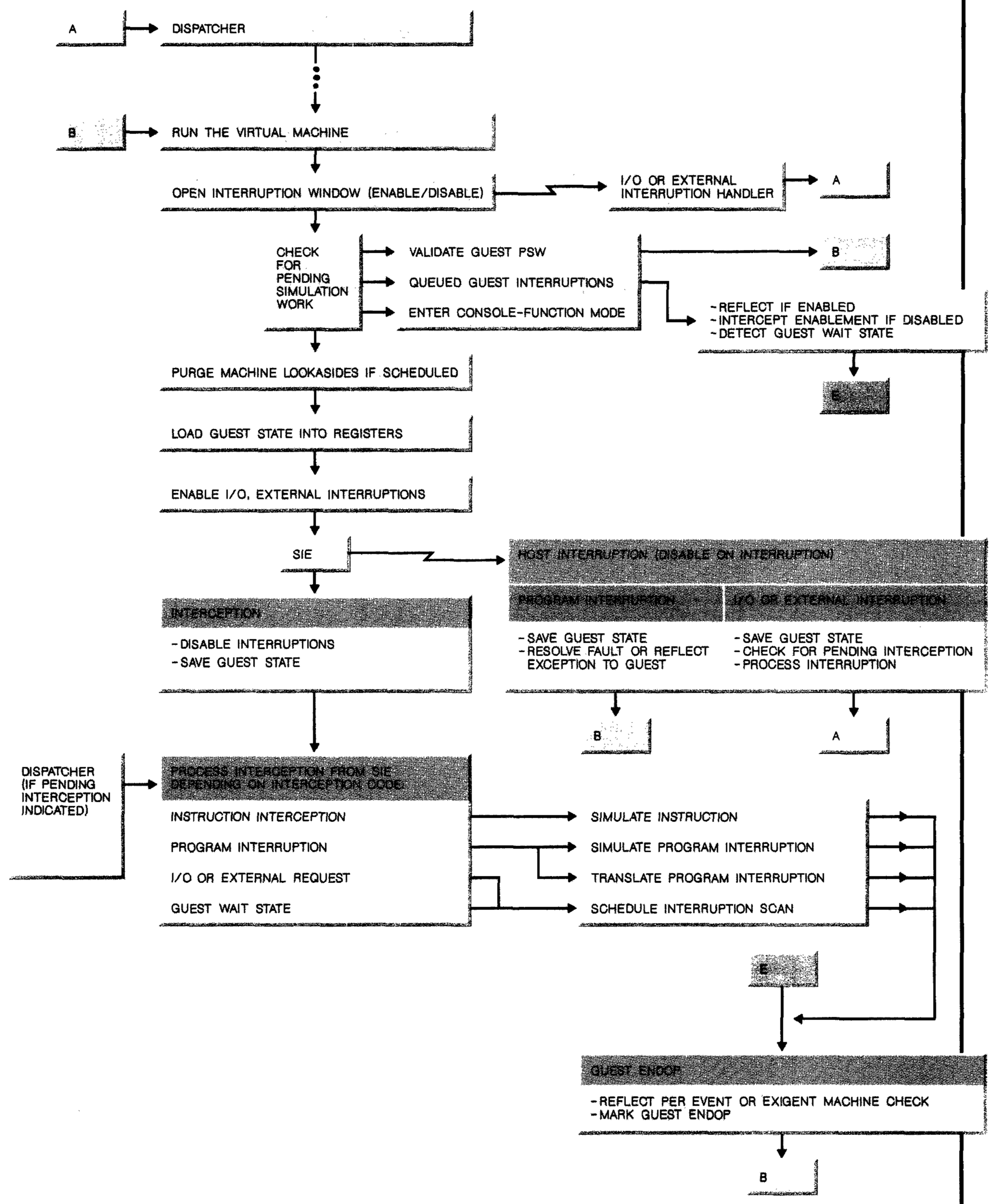
Use of interpretive execution in VM/ESA

Structure of virtual-machine simulation in CP. Figure 3 depicts the flow of control through the virtual-machine simulator in CP. Before actually issuing the SIE instruction, the simulator must accept pending host interruptions, attend to scheduled simulation functions, ensure that machine lookaside buffers are up-to-date, and load the guest state. When SIE finishes, the simulator saves the guest state and processes the condition reported by the machine. Each of these steps is explained in more detail below. Where particular operations reflect the idiosyncrasies of CP, background on the structure of CP is included as well.

Host interruption handling. When the dispatcher in CP passes control to the simulator to run a virtual CPU, the simulator first opens a *window* for I/O and external interruptions. That is, it issues instructions to enable and immediately thereafter to disable these interruptions. This ensures that all currently pending interruptions are processed before the simulator continues. Such interruptions may represent higher-priority work to take precedence over simulation, or the release of a resource (such as an I/O device) for which the next operation should be started, to maximize utilization.

CP takes an unusual approach to asynchronous interruptions, such as I/O or external interruptions. (*I/O interruptions* indicate completion of an operation by the channel subsystem. *External interruptions* indicate signals from other host CPUs, expiration of a timeslice, or arrival at a desired real-time value.) Most CP code runs disabled for these interruptions. Logic throughout CP takes advantage of this nonpreemptive dispatching model to simplify interprocess serialization. CP's dispatcher and virtual-machine simulator open interruption windows to accept pending interruptions at points when a *loss of control* (the cessa-

Figure 3 Virtual-machine simulation in CP



A
DISPATCHER
B
RUN THE VIRTUAL MACHINE
OPEN INTERRUPTION WINDOW (ENABLE/DISABLE)
I/O OR EXTERNAL INTERRUPTION HANDLER
A
CHECK FOR PENDING SIMULATION WORK
VALIDATE GUEST PSW
QUEUED GUEST INTERRUPTIONS
ENTER CONSOLE-FUNCTION MODE
B
-REFLECT IF ENABLED
-INTERCEPT ENABLEMENT IF DISABLED
-DETECT GUEST WAIT STATE
E
PURGE MACHINE LOOKASIDES IF SCHEDULED
LOAD GUEST STATE INTO REGISTERS
ENABLE I/O, EXTERNAL INTERRUPTIONS
SIE
HOST INTERRUPTION (DISABLE ON INTERRUPTION)
PROGRAM INTERRUPTION
I/O OR EXTERNAL INTERRUPTION
-SAVE GUEST STATE
-RESOLVE FAULT OR REFLECT EXCEPTION TO GUEST
-SAVE GUEST STATE
-CHECK FOR PENDING INTERCEPTION
-PROCESS INTERRUPTION
B
A
INTERCEPTION
-DISABLE INTERRUPTIONS
-SAVE GUEST STATE
DISPATCHER (IF PENDING INTERCEPTION INDICATED)
PROCESS INTERCEPTION FROM SIE DEPENDING ON INTERCEPTION CODE:
INSTRUCTION INTERCEPTION
PROGRAM INTERRUPTION
I/O OR EXTERNAL REQUEST
GUEST WAIT STATE
SIMULATE INSTRUCTION
SIMULATE PROGRAM INTERRUPTION
TRANSLATE PROGRAM INTERRUPTION
SCHEDULE INTERRUPTION SCAN
E
GUEST ENDOP
-REFLECT PER EVENT OR EXIGENT MACHINE CHECK
-MARK GUEST ENDOP
B

tion of the executing task and the dispatching of another) is tolerable; in addition, the simulator allows interruptions during the time that the guest runs in interpretive-execution mode.

CP's interruption handlers generally do not resume execution at the point of interruption, as do those in most operating systems. Rather, control is returned to the dispatcher, which then passes control to a routine which has been scheduled for execution. For example, following an interruption in the simulator, the dispatcher will eventually return control to the *front end* of the simulator, causing any logic before the point of interruption to be re-executed. The dispatcher and simulator are structured such that, if an interruption occurs during their window, re-entry from the beginning is not harmful. In fact, in certain cases, it is essential that the front-end code be re-executed, because the task dispatched on an interruption may modify data on which the interrupted task depends.

When the guest's state is loaded and the machine is enabled in preparation for interpretive execution, an indicator is set. Interruption handlers will recognize this indicator and save the guest state before proceeding. Control will later come through the front end of the simulator, past preliminary tests, to the point at which the guest's state is reloaded and its execution is resumed.

Pending interruptions processed during interruption windows need not save the state of the interrupted task, since control will not be returned to the point of interruption. This reduces the cost of handling interruptions.

Verifying guest endop. When the dispatcher is entered and finds no higher-priority CP work for a given virtual machine, it ordinarily passes control to the simulator. Therefore, the simulator may be re-entered at any loss of control, such as a page fault during simulation of an instruction. The simulator recognizes this re-entry and terminates processing, signaling the dispatcher that it should not re-enter the simulator for this virtual CPU. This prevents performing another simulation operation or starting interpretive execution while the previous operation is in progress.

If the simulator finds no indication of work in progress, it declares the virtual machine to be at *endop*, i.e., at the end of a guest unit of operation. (As defined in Reference 9, a *unit of operation* is the execution of either a single ordinary instruction or a portion of an interruptible instruction. Interruptions may occur only between units of operation.)

Pending simulation work. Once the simulator has opened and closed its initial interruption window, it checks for specific work requests. Certain functions may be scheduled either by a simulated guest operation or asynchronously, and must be performed before guest execution can be resumed. Examples of such operations are: checking a new guest program status word (PSW) introduced by simulation, scanning for virtual interruptions to be reflected, and entering "console-function mode."

Each of these is described in more detail below.

Checking the guest PSW—Simulation of guest interruptions and certain instructions introduces new information into the guest PSW. In these cases, CP must check the validity of the PSW before resuming guest execution. This ensures that program interruption conditions due to PSW format errors are recognized at the proper priority level.

Scanning for guest interruptions—CP queues all I/O interruptions for ordinary (not high-performance) guests and simulated external interruptions for VM communication functions until the guest reaches a point at which they can be reflected. When an interruption is queued, an "interruption scan" is scheduled, and is performed at the next "guest endop," i.e., the end of the guest instruction (whether executed by the machine or simulated by CP). The structure of the simulator ensures that control reaches this point only at guest endop.

At that time, if the enablement masks in the guest PSW and control register permit the interruption, it is dequeued and presented; if not, intervention or interception controls are set in the state description, so that an interception will occur when the guest changes the relevant mask in the PSW or control register. On such an interception, a new interruption scan is scheduled and performed to try again to present the interruption.

If no interruptions are reflected, the scan ends by checking for a guest PSW specifying wait state. A wait-state guest is generally not permitted to enter interpretive execution. Rather, simulation of the guest is suspended until a guest interruption arrives and is reflected, introducing a run-state PSW.

Entering console-function mode—Console-function mode (CFM) is the environment in which CP console commands are processed. Since many commands inspect or alter the virtual-machine state, they must not be allowed to execute concurrently with simulation work. If a request to switch to CFM arrives and the virtual machine is not at endop, a CFM entry is scheduled for the next guest endop.

VM/ESA uses two means of purging lookaside buffers.

Purging machine lookaside buffers. Interpreting a pageable guest implicitly invokes host DAT, and interpreting VM Data Spaces mode guest additionally invokes host ART. These machine processes can internally buffer information from the host translation tables and use the buffered information to avoid refetching from host tables. The term *translation-lookaside buffer* (TLB) refers to the internal copies of DAT-table entries, and *ART-lookaside buffer* (ALB) refers to internal copies of ART-table entries. When CP changes its translation tables, it must instruct the machine to purge its lookaside buffers in order to prevent obsolete information from being used in subsequent translations.

VM/ESA uses two means of purging lookaside buffers: an INVALIDATE PAGE TABLE ENTRY (IPTE) instruction, which broadcasts the order to all processors, and PURGE TLB and PURGE ALB instructions, which purge a single processor's buffers. To reduce overhead, CP often chooses to invalidate batches of PTEs directly (without IPTE) when an immediate broadcast is not required. In these cases, CP keeps a record, in the form of timestamps, of the need to purge the lookaside buffers before running a guest which might access the affected virtual storage. For each processor, CP keeps the time of the last purge on that processor. (CP always purges the TLB and ALB in tandem.) For each virtual machine, CP keeps the time of the last change to a translation-table entry which necessitated a purge.

Before starting interpretive execution, the simulator compares the virtual machine's purge-required timestamp with that of the most recent actual purge on the current processor. If the processor has been purged recently enough (e.g., in preparing to run a different virtual machine), then it does not need to be purged again. Since this check is made prior to every SIE instruction, as the virtual machine is dispatched successively on different processors, each processor's buffer will be purged when and only when necessary.

This timestamp checking is one of the operations which must be re-executed following every loss of control, because a new purge-required timestamp may be set during the loss of control. (CP's storage management will not take any action which demands a purge while this virtual machine is actually dispatched, but may do so when it loses control.) Re-entering the simulator at the beginning following a loss of control ensures that the check will be repeated.

Issuing SIE. Once these preliminaries are handled, the simulator can turn to the business at hand: starting interpretive execution of the guest. The simulator loads the guest state into the appropriate registers, enables host interruptions, and issues START INTERPRETIVE EXECUTION (SIE) to turn over guest execution to the machine. The machine remains in interpretive-execution mode, executing guest instructions and interruptions, until either a host interruption occurs or the machine intercepts a guest event which requires host attention.

If a synchronous host interruption such as a host page fault occurs, then the guest state is saved and the condition handled. Faults are usually handled by paging in the needed data, updating translation-table entries, and re-entering the simulator to resume guest execution. Certain host program interruptions indicate improper guest operations; these are turned into the appropriate guest program interruptions. For example, a fault arising from a reference to a location not in the guest's addressable range is rendered as a guest addressing exception.

If an asynchronous host interruption such as an I/O or external interruption occurs, the guest state is saved. CP then checks for a *pending interception*. This might occur if the machine detected and presented an interception and completed the SIE instruction, but a host interruption then occurred before CP could disable interruptions. CP recognizes that an interception was reported in the state description and schedules handling of the interception at higher priority than normal re-entry to the simulator. At this point, the guest's state has been saved and CP's interruption handler can process the interruption. Eventually the interruption handler will exit to the dispatcher.

If an ordinary interception (without host interruption) occurs, the machine resumes execution of the simulator immediately after the SIE instruction. The simulator disables interruptions, saves the guest state, and goes on to process the interception.

Interception processing. When an interception occurs, the machine stores into the state description a code identifying the type of condition which occurred and additional information depending on the condition, such as the text of an intercepted instruction or the parameters associated with an intercepted guest interruption. The simulator then acts on the condition.

Instruction interception—If an instruction is intercepted, the proper simulation routine receives control, according to the operation code. Exception conditions are detected and presented in the form of guest program interruptions. Otherwise, the simulation is completed and guest endop is declared.

CP sometimes requests interception of instructions which the machine is capable of interpreting. In these cases, the simulation routine may include special handling for the case for which interception was requested. For example, when the interruption scan finds a guest I/O interruption which must be held pending because the corresponding interruption subclass is disabled in the guest control register, it requests interception of LOAD CONTROL (LCTL) instructions which alter that control register. The LCTL simulation routine schedules a fresh interruption scan, so that the interruption can be presented if it is enabled according to the new control-register value.

Guest program interruption—Most guest program interruptions are presented directly by the machine. However, certain conditions are always intercepted to permit special processing by the host. Some interruption conditions under VM Data Spaces are reported in one way by the machine, but are to be reflected to the ESA/XC virtual machine in another form. For example:

- The machine reports specific exceptions for failures at each step in host ART. These either drive host recovery or are mapped into the application-oriented exceptions of ESA/XC.
- To simplify machine implementation, an instruction which depends on DAT gives a special-operation exception under VM Data Spaces, just as it does when DAT is off in ESA/390. CP translates such an exception into an operation exception, signifying that the instruction does not exist at all in ESA/XC.

I/O and external intervention requests—As described earlier, when the CP's interruption scan finds a pending guest I/O or external interruption which is disabled in the guest PSW, it uses the intervention controls in the state description to request interception when the PSW becomes enabled. On receiving such an "external-request" or "I/O-request" interception, the simulator schedules an interruption scan to present the pending interruption.

Guest wait state—When the machine encounters a guest PSW designating wait state, it returns control to the host, and the simulator schedules an interruption scan. If no interruption can be presented, the guest is marked "not ready" and a timer event is scheduled to redispatch the guest when any of its virtual timers would present an interruption. This is the only case in which CP must manage guest timers. As long as the guest remains ready, the CP can rely on the machine to update the guest timers and recognize the associated interruptions when the guest is dispatched.

Guest endop. All instruction and interruption simulations converge at a common point to conclude the virtual unit of operation. Here, CP reflects any guest program-event-recording (PER) interruption conditions that were recognized either by the machine on instruction interception or by CP during instruction simulation.

In addition, simulation may encounter host problems, such as logic errors or loss of guest data on auxiliary storage. These problems may have corrupted the guest state or the execution of the guest program. Such conditions are reported to the guest as exigent machine checks. For example, loss of a page of host virtual storage is effectively loss of data in guest absolute storage, and is reported to the guest as an uncorrected storage error. The guest program then has the opportunity to recover from that condition.

The architecture requires that these PER events and exigent machine checks occur synchronously with the end of the unit of operation. Therefore, CP's simulator ensures that these events are presented before guest endop is declared. (Only after endop is declared can further simulation, such as presentation of an asynchronous interruption or execution of another guest instruction, proceed.)

Of course, in the vast majority of cases, there is no PER event or machine check to present, and endop is declared immediately. After endop is declared, control is returned to the top of the simulator, which performs any scheduled simulation work (such as an interruption scan) or resumes guest execution.

Virtual SIE. An interesting perspective on interpretive execution is the simulation of the SIE instruction itself, when that instruction is issued by a guest.

As mentioned above, some machine models can interpret a "second-level" SIE instruction issued by a high-performance guest. Such a function is a tremendous boon to migration, allowing an orderly, cautious upgrade from one release of VM to the next. However, the ability to run a second-level VM system as an ordinary guest for testing and debugging, without dedicating real resources as high-performance guests demand, has always been an important feature of VM. Customers have come to depend on this for first-shift VM systems programming, testing of software fixes, and so forth. Moreover, not all models support Region Relocation and Interpreted SIE. For these cases, CP's simulation software must fill the gap, in a function known as interpretive-execution simulation or more simply "virtual SIE" (vSIE).[12]

When the machine does not interpret a second-level SIE, the guest SIE instruction yields an instruction interception. CP passes control to an instruction simulation routine as for any other instruction interception. However, simulating the entire execution of the guest SIE, including execution of all the second-level guest's instructions

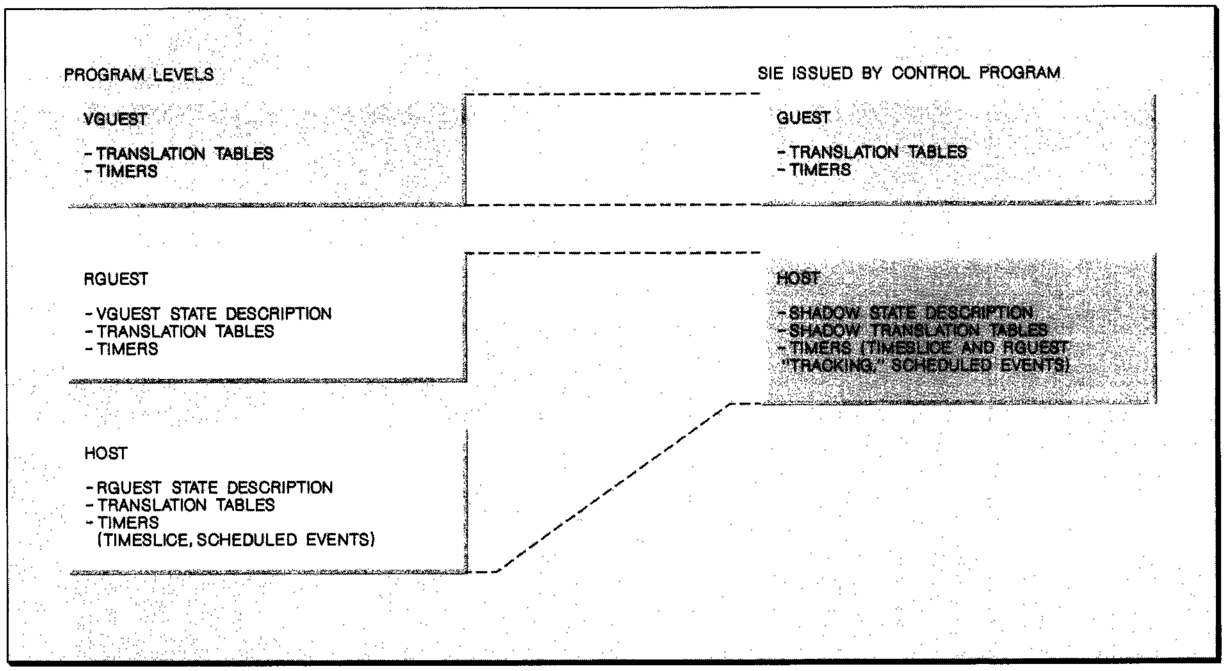
CP enlists the machine's aid, by using SIE itself to simulate SIE.

and interruptions, is not practical. VM would have to pay a huge, continuing cost to maintain simulation routines for every instruction in the architecture, and such simulation would perform abominably. Rather, CP enlists the machine's aid, by using SIE itself to simulate SIE.

The approach taken is to use a SIE instruction at the host level to cause the machine to interpretively execute the second-level guest. However, CP must make adjustments to accommodate the additional "level" of functions. As described earlier, SIE recognizes two levels of program, host and guest; it can perform two levels of address translation, manage two levels of timing facilities, and so on. In the most complex case, vSIE must effect three levels of address translation and support three sets of timers—for the host, the first-level guest (called the "real guest" or RGuest), and the second-level guest (the "virtual guest" or VGuest). In order to have the machine interpret the VGuest, these three levels must be collapsed into two. As depicted in Figure 4, CP lets the machine see the VGuest as "the guest," and presents an image of "the host" which merges CP's actual host view with the RGuest's view.

In using SIE to simulate SIE, VM/ESA resorts to the time-honored virtualization technique of shadowing. First, the state description itself must be shadowed. CP's vSIE support copies the state description specified as the operand of the RGuest SIE instruction and edits it to represent the host's view of the VGuest, for example, by translating RGuest real addresses of control structures into the host real addresses expected by the machine. Then, vSIE must create shadow translation tables

Figure 4 Mapping of host and guest levels under vSIE


PROGRAM LEVELS
SIE ISSUED BY CONTROL PROGRAM
VGUEST
- TRANSLATION TABLES
- TIMERS
GUEST
- TRANSLATION TABLES
- TIMERS
RGUEST
- VGUEST STATE DESCRIPTION
- TRANSLATION TABLES
- TIMERS
HOST
- SHADOW STATE DESCRIPTION
- SHADOW TRANSLATION TABLES
- TIMERS (TIMESLICE AND RGUEST "TRACKING," SCHEDULED EVENTS)
HOST
- RGUEST STATE DESCRIPTION
- TRANSLATION TABLES
- TIMERS (TIMESLICE, SCHEDULED EVENTS)

to apply the composition of RGuest and host address translation. These tables will be presented to the machine as host translation tables, so that they will apply after VGuest DAT and prefixing (if any).

Once the shadow structures are built, CP issues SIE against the shadow state description. Most interception conditions can be reflected directly as interceptions on the RGuest SIE instruction; some require special processing in CP. Whenever an interception or interruption is to be presented to the RGuest, the contents of the shadow state description, representing the current VGuest state, must be transcribed into the state description in RGuest storage.

During VGuest execution, the machine may present translation exceptions to the host as it encounters entries marked invalid in the shadow translation tables. The host must then use the contents of RGuest tables along with the results of translation at the host level to update the shadow tables. In the process, it may be necessary to page in RGuest storage or to reflect to the RGuest an exception detected in its tables.

RGuest timing facilities must also be simulated through host timers, since the machine can support only two levels of timing, the VGuest's and the host's. Before issuing SIE on the shadow state description, CP sets the host CPU timer (which ordinarily holds the dispatching timeslice) to the minimum of the timeslice and the time remaining until the next RGuest CPU-timer or clock-comparator interruption is due. When the host interruption occurs, the CP will either recognize timeslice end or present the interruption to the RGuest, as appropriate.

Concluding remarks

Although interpretive-execution architecture began as a test and migration aid for native architecture developments, it has become a platform for extending the utility of virtual machines. Some of the developments in the evolution of this architecture have been described in this paper, including the ability to run concurrent production-level virtual machines. These developments primarily dealt with the interpretation of a native architecture. The latest development, VM Data Spaces, which has been described in detail, breaks some of the otherwise close ties to real components. VM Data Spaces establishes the interpretive-execution architecture as a platform for innovations unique to the virtual-machine environment as well as those common to native architecture developments.

Effective use of interpretive execution requires close cooperation between the machine and a host program like VM/ESA CP. CP fills the gaps in interpretive execution, so that the guest sees a complete, architecturally consistent machine interface. Interpretive execution provides a flexible structure whereby both hardware and software can be brought to bear on a problem, and each can contribute its own strengths to the solution.

Cited references and notes

1. P. H. Gum, "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM Journal of Research and Development* **27**, No. 6, 530–543 (1983).
2. *IBM System/370 Extended Architecture: Interpretive Execution*, SA22-7095, IBM Corporation; available through IBM branch offices.
3. J. M. Gdaniec and J. P. Hennessy, "VM Data Spaces and ESA/XC Facilities," *IBM Systems Journal* **30**, No. 1, 14–33 (1991, this issue).
4. The information in this paper pertains only to the ESA feature of VM/ESA Release 1.0 and to VM/ESA Release 1.1. Since the interpretive-execution facility does not exist in System/370 architecture, the 370 feature of VM/ESA Release 1.0 cannot make use of it.
5. In common VM parlance, "guest" refers specifically to a second-level System Control Program, such as MVS/ESA or VM/ESA, as opposed to a CMS virtual machine. The interpretive-execution architecture extends the term to apply to any program running in a virtual machine.
6. ESA/390 defines three levels of storage address: Dynamic address translation transforms a *virtual* address into a *real* address. Prefixing is performed on a real address to yield an *absolute* address, which designates a location in physical storage. Prefixing uses the contents of the CPU's prefix register to "swap" addresses 0–4095 with another address range, so that each CPU can have access to different low storage for interruption parameters, save areas, and processor-specific data.
7. Dynamic address translation (DAT) converts a virtual address to a real address by means of a selected entry in the segment table, and a selected entry in the page table designated by that segment-table entry. The result is the real address of the storage frame corresponding to the virtual address.
8. T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk, "Multiple Operating Systems on One Processor Complex," *IBM Systems Journal* **28**, No. 1, 104–123 (1989).
9. *IBM Enterprise Systems Architecture/370: Principles of Operation*, SA22-7200, IBM Corporation; available through IBM branch offices.
10. K. E. Plambeck, "Concepts of Enterprise Systems Architecture/370," *IBM Systems Journal* **28**, No. 1, 39–61 (1989).
11. Although ESA/390's address-space numbers (ASNs) and the ASN translation structures (ASN first tables and ASN second tables) do not exist in VM Data Spaces mode, VM Data Spaces retains the ASN-second-table entry (ASTE) as the control structure that defines an address space.
12. P. H. Tallman, *Instruction Processing in Higher Level Virtual Machines by a Real Machine*, U.S. Patent No. 4,792,895.

Damian L. Osisek *IBM Data Systems Division, P.O. Box 6, Endicott, New York 13760.* Mr. Osisek is an advisory programmer in VM/ESA design at the IBM Endicott Programming Laboratory. He joined IBM in 1983 as a programmer in VM/XA development and worked in the areas of virtual-CPU simulation and real-CPU management. He has designed and developed portions of VM/XA's support for the 3090 vector facility and PR/SM and VM/ESA's simulation of the VM Data Spaces facility in virtual SIE. Mr. Osisek earned a patent for "Passive Serialization in a Multitasking Environment." He received a B.A. degree in computer science and classics from Rutgers University in 1981 and an M.S. degree in computer science from Rensselaer Polytechnic Institute in 1987.

Kathryn M. Jackson *IBM Data Systems Division, P.O. Box 950, Poughkeepsie, New York 12602.* Ms. Jackson is a staff programmer in the Enterprise Systems Central Architecture department in IBM's Poughkeepsie Development Laboratory. She received a B.S. degree in computer science and mathematics from Hofstra University in 1982 and joined IBM as a programmer in the Data Systems Assurance organization. In 1988, she joined Enterprise Systems Central Architecture where she participated in the design of several extensions to the interpretive-execution architecture, including the VM Data Spaces facility.

Peter H. Gum *IBM Data Systems Division, P.O. Box 950, Poughkeepsie, New York 12602.* Mr. Gum is a Senior Technical Staff Member and a member of the Enterprise Systems Central Architecture department. He joined IBM in 1964 in Poughkeepsie as a system programmer working on the operating system for the IBM System/360, and subsequently participated in the design of several versions of the control program. In 1973 he joined the architecture department, where he participated in the design of extensions to the architecture of the IBM System/370. He received a B.A. from Oberlin College, Oberlin, Ohio, in 1958 and an M.A. from the American University, Washington, D.C., in 1962, both in mathematics. Mr. Gum is a member of the Association for Computing Machinery.

Reprint Order No. G321-5422.