VM Data Spaces and ESA/XC facilities

by J. M. Gdaniec
J. P. Hennessy

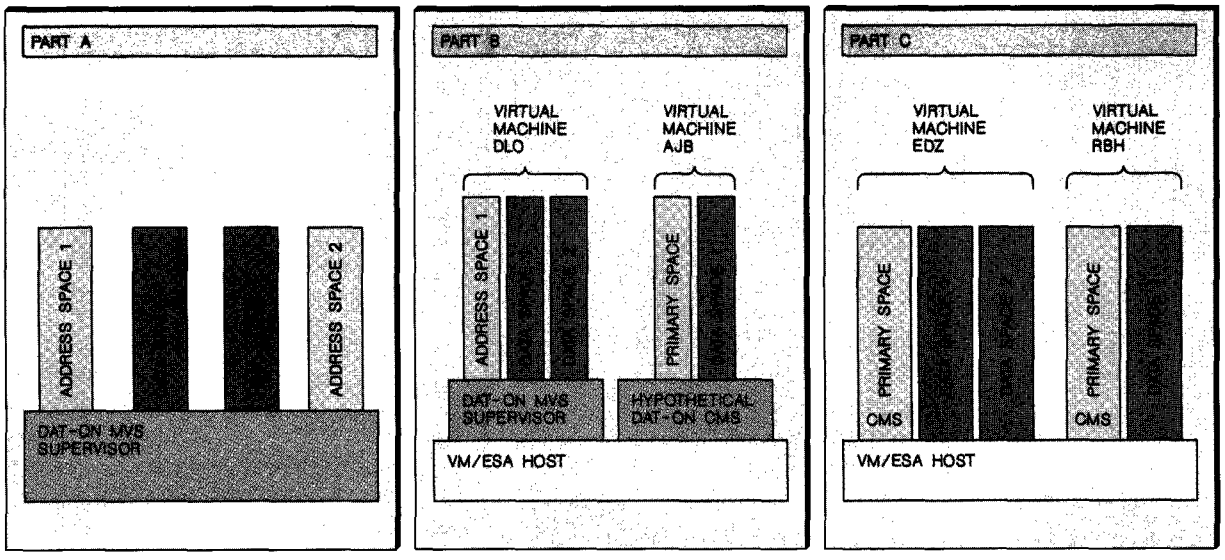
Release 1.1 of the Virtual Machine/Enterprise Systems Architecture™ (VM/ESA™) operating system introduces a new function called VM Data Spaces, provided through a new virtual-machine architecture called Enterprise Systems Architecture/Extended Configuration (ESA/XC). ESA/XC is the strategic VM/ESA virtual-machine environment for Conversational Monitor System (CMS) users and service virtual machines requiring large amounts of storage or advanced data-sharing capabilities. ESA/XC includes all of the facilities of System/370 Extended Architecture (370-XA) that are used by CMS or server programs and is therefore upward compatible for CMS or server programs currently running in 370-XA virtual machines. To this 370-XA base, ESA/XC adds the data space and access-register addressing capabilities previously available only under the Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA™) operating system. These addressing extensions can be used to make additional storage available to large, storage-constrained applications and can also be used by servers as an efficient way of sharing data between service virtual machines and the users that access those servers. As an introduction to the VM Data Spaces function, this paper describes the ESA/XC virtual-machine architecture and presents an overview of the VM/ESA services provided in support of the ESA/XC architecture.

The IBM Enterprise Systems Architecture/370™[1] introduced advanced address-space facilities that are exploited by supervisor programs to provide application programs with the ability to access many address spaces concurrently. These facilities are also provided by the new Enterprise Systems Architecture/390™ (ESA/390™).[2] The Enterprise Systems Architecture/Extended Configuration (ESA/XC) is a new virtual-machine architecture that brings this powerful capability to the Virtual Machine/Enterprise Systems Architecture™ (VM/ESA™) user in a form suitable to the Conversational Monitor System (CMS) virtual-machine environment. This paper serves as an introduction to ESA/XC, describing some of the important features of the architecture, its basic foundation, and its useful services, and providing a glimpse of the philosophy underlying the architecture.

An address space is simply a sequence of addresses that starts at zero and proceeds up to a value that varies according to the size of the address space. Each address designates a byte of information. For example, when a CMS user logs on to the system, CP (the control program) creates for that user a virtual machine that includes, among other things, "storage." Although the CMS user may not generally think of it in this way, that storage is an address space. To an application program, an address space is storage and is used to hold data. The nuances of storage management that are important to the underlying supervisor or host are not important to the application program.

The advanced address-space facilities of ESA/390 provide a supervisor with the means by which it can create many address spaces and selectively make these address spaces accessible to applica-

Figure 1 Comparison of ESA/390 address spaces and ESA/XC address spaces



tion programs. Suitably coded application programs can then enter a mode called the access-register mode to directly address data in these address spaces. Application programs that share a common supervisor can even share these address spaces, thus exchanging information in a very efficient manner.

The intuitive approach for making these powerful abilities available to the CMS application program might call for enhancing CMS to run in an ESA/390 virtual machine, having it build and maintain address spaces. Figure 1 compares address spaces built by operating systems using ESA/390 (Part A) with address spaces built by a hypothetical CMS using ESA/390 (Part B).

In this form, however, ESA/390 would be difficult for CMS to exploit. Consider the following:

- CMS is a single-user supervisor. Since two CMS users do not share the same supervisor, they cannot run application programs that share the same address spaces. The address spaces created by one virtual machine using ESA/390 are isolated from all other virtual machines. One of the goals of VM/ESA is to improve the facilities available for interuser communication, but CMS exploitation of ESA/390 would not promote this. To allow direct data sharing between virtual machines, another approach is necessary.
- To use the access-register mode in ESA/390, an application program must run with dynamic address translation (DAT) on, something that is not supported today by CMS. Although it would be possible to change CMS to run with translation on, this would undoubtedly have a negative effect on some existing application programs. CMS runs application programs in the supervisor state, which gives application programs full access to all virtual-machine facilities. Many application programs take advantage of this situation by performing their own manipulations of the virtual-machine environment. Since it is difficult to code such programs so as to anticipate future changes in CMS, it is likely that at least some application programs would cease to function correctly if CMS were to start running with translation active.
- A supervisor that builds address spaces must be fairly sophisticated. To manage storage efficiently, a supervisor should keep track of how frequently data in an address space are used and should "page out" data that are infrequently accessed. A supervisor that performs such paging must then keep track of where these data reside on auxiliary storage devices such as a direct-access storage device (DASD) or expanded storage, and be prepared to read them in again when they are needed by an application program. Sophisticated storage-management techniques such as these require a great deal of

code and would thus significantly increase the size of the CMS nucleus. Worse than that, since CP pages the host address space representing guest storage, two levels of paging activity would be taking place if CMS also paged its address spaces. Furthermore, because CMS relies on CP to be the system resource manager, it does not today need to manage auxiliary storage devices such as paging DASD, and thus does not have access to any.
- CP already contains logic to manage address spaces. It was mentioned earlier that CP creates an address space for a user when that user logs on to the VM/ESA system. Enhancing this logic in CP is preferable to duplicating it in CMS.

The ESA/XC architecture, a derivative of ESA/390, has been developed and tailored specifically for the virtual-machine environment. A cooperative effort between VM/ESA and the machine[3] allows VM/ESA to provide a virtual-machine architecture that better satisfies CMS application-program requirements than architectures available on a real machine. Significantly, no real machine can operate in this architecture mode; only virtual machines use this architecture.

ESA/XC is derived from ESA/390.

Highlights of ESA/XC

The ESA/XC architecture is derived from the ESA/390 architecture, which itself has evolved from System/370 Extended Architecture (370-XA). Someone familiar with the basic elements of 370-XA or ESA/390 should have no trouble becoming familiar with the ESA/XC architecture. In fact, because it is an architecture intended for application programs of the type that run under CMS, it is actually simpler than either 370-XA or ESA/390.

Relationship of ESA/XC to ESA/390. As mentioned previously, in ESA/390, an application program must run with dynamic address translation (DAT) active in order to enter the access-register mode to reference address spaces. This condition is not the case in the ESA/XC architecture, and in fact, the DAT facility does not even exist in ESA/XC. The many structures described in the ESA/390 architecture for the management of DAT, such as DAT tables, ASN translation tables, PC-number translation tables, linkage tables, and entry tables, do not therefore exist in ESA/XC.

The interpretive-execution facility is also not provided in the ESA/XC architecture. This facility, which is invoked by the START INTERPRETIVE EXECUTION instruction, is intended to be used by a host for the emulation of virtual machines. The ESA/XC architecture is intended for application programs, not hosts like VM/ESA, so the interpretive-execution facility is not provided in ESA/XC.

Several control instructions are defined in ESA/390 for the use and management of the facilities that do not exist in ESA/XC. Since they are not needed, these instructions do not appear in ESA/XC. Some of these instructions are BRANCH AND STACK (BAKR), PROGRAM CALL (PC), PROGRAM RETURN (PR), LOAD REAL ADDRESS (LRA), and START INTERPRETIVE EXECUTION (SIE).

All other features of ESA/390 and 370-XA continue to exist in ESA/XC. For example, I/O operations are performed using the same instruction set as in ESA/390 and 370-XA, programs still have the choice between running in 24-bit addressing mode and 31-bit addressing mode, and access registers are still used to control direct references to address spaces. As far as normal CMS application programs are concerned, the ESA/XC architecture is upward compatible with ESA/390.

Basic elements of ESA/XC. The advantage of ESA/XC for a CMS application program is apparent in the new virtual-machine services provided in ESA/XC. These services will be described in detail later but are introduced here. They are provided to:

- Create, share, and otherwise manage address spaces
- Gain and relinquish access to an address space
- Map data on DASD to pages of an address space, allowing a program to subsequently reference data on DASD by instead referencing the address space

Before describing these services in detail, it is necessary to describe some of the basic concepts related to address spaces in ESA/XC.

Address spaces. Every virtual machine running in the ESA/XC architecture owns at least one address space, the primary address space, given to the user by CP when the user logs on to the VM/ESA system. The size of this address space is determined from the entry describing that user in the user directory, or from a subsequent DEFINE STORAGE command. After logging on, if authorized in the user directory, a user may create other address spaces, and if desired, share them with other logged-on users. Address spaces created by a program in this manner are sometimes called data spaces. A data space exists until it is either explicitly destroyed by the owner or until the owning virtual machine goes through a virtual-machine-reset operation. Note that a virtual-machine-reset operation occurs implicitly during the processing of such commands as IPL (initial program load) and LOGOFF.

A (DAT-off) CMS program using ESA/XC can obtain concurrent access to more storage than a CMS program using 370-XA or ESA/390. More storage is available because an authorized program using ESA/XC can create many address spaces, each of which may be up to two gigabytes (2G) in size. Thus, a DAT-off program using 370-XA or ESA/390 can address at most 2G of storage, but a program using ESA/XC can concurrently address more than 2G.

Figure 1 compares address spaces built by programs using ESA/390 with address spaces built by programs using ESA/XC. Part A shows address spaces built by an operating system that exploits ESA/390. Part B shows those address spaces alongside address spaces built by a hypothetical CMS that exploits ESA/390. Note that such address spaces would not be shareable between virtual machines. Part C shows address spaces built by application programs on CMS in virtual machines using ESA/XC. Address spaces built by ESA/390 virtual machines are *guest* address spaces managed by the virtual-machine supervisor, whereas address spaces built by ESA/XC virtual machines are *host* address spaces managed by CP.

Every ESA/XC address space has the following attributes:

- *A name*—An address-space name is a string from one to 24 characters long. Except for the primary address space of the virtual machine, the address-space name is assigned by the owning program when the address space is created, and never changes. CP assigns the name BASE to the primary address space of the virtual machine. The full address-space identification of an address space consists of a string up to 33 characters long and is composed of the owning userid (the userid that created the address space), a colon, and the name. Since no two address spaces owned by the same user are permitted to have the same name, and no two users logged on to the same system are permitted to have the same userid, no two address spaces in the system can have the same address-space identification. Note, however, that nothing prevents a program from destroying an address space and then creating another with the same name.

- *An address-space-identification token*—The 33-character address-space identification described above is designed for easy interpretation by VM/ESA users and is somewhat unwieldy to manipulate in a program. When an address space is created, therefore, CP assigns to the address space an eight-byte token called the address-space-identification token, or ASIT. This token is returned to the owner of the address space when the address space is created. The ASIT has these important characteristics:

  1. It is a system-wide unique value. That is, no two address spaces in the system, including those created by other users, will have the same ASIT.
  2. It is nonrepeating for the life of the current system IPL. In other words, once an ASIT is assigned to an address space, that same ASIT will never be assigned to another address space, even after the address space to which it was first assigned is destroyed.[4]
  3. An address space has only one ASIT associated with it. Every user refers to the same address space using the same ASIT. ASITs that are equal designate the same address space; ASITs that are not equal designate different address spaces.

The combination of these characteristics allows an application program to use the ASIT as the principal identification of an address space.

It is interesting to note that with these guarantees, the ASIT associated with the primary address space of a virtual machine can be used as a unique designation of a particular instance of a particular userid being logged on to the system. That is, if a user logs off and then back on, his or her userid is the same, but the ASIT corresponding to his or her primary address space is different. An application program that was to keep track of such things might therefore use the primary ASIT instead of the userid to identify the virtual machine.

- *Size*—Within the bounds imposed upon it in the user directory, the creating program chooses the size of the address space, giving it any size between 64 kilobytes and two gigabytes. CP will round the size of the address space up to the nearest 64-kilobyte boundary.

- *Private or shareable status*—When an address space is first created, it is a private address space, accessible only to the owning virtual machine. Similarly, when a user first logs on, the primary address space of the virtual machine is private. If the owning program chooses, and if it is authorized to do so, it can grant another virtual machine permission to access an address space that it owns, including its primary address space. The address space is then shareable and remains so until it is destroyed, until the owner goes through a virtual-machine-reset operation, or until the owner requests that the address space be again isolated from other virtual machines.

- *List of permitted users*—Every ESA/XC address space is accessible to some set of users on the system. A private address space is accessible to only the owning user. Shareable address spaces are accessible to those users who have been granted permission by the owner to access the address space. The owner always has read-write permission to an address space created by the owner. Other users have either read-only access or read-write access to the address space, depending on what was granted them by the owner. The owner may grant different users different types of permission. There is no limit to the number of users that may be granted permission, and therefore, there is no limit to the number of users that may concurrently access the address space.

Access lists. Before a program can actually read or write data in a nonprimary address space, it must invoke a CP service to add an entry designating that address space to its access list. Each virtual configuration has its own access list, whose entries determine which address spaces the virtual CPUs in that configuration can reference at any one time. The number of entries in the access list varies between six and 1022 and is controlled by information in the user's directory entry. The default size is six.

When a program invokes the VM/ESA service to add an address space to its access list, CP selects an unused entry in the access list, fills it in as requested by the program, and returns a four-byte access-list-entry token (ALET) to the program. As described later, a program uses this ALET to make direct references to the address space. The access-list entry thus allocated remains allocated until the program explicitly removes the entry, or until the virtual machine goes through a virtual-machine-reset operation.

Every ESA/XC access-list entry has the following attributes:

- *A state*—An access-list entry is always in one of the following states:

  An *unused* access-list entry (ALE) is one that has never been allocated, one that has become deallocated by a Remove ALE operation of the virtual machine, or one that has become deallocated because the virtual machine went through a virtual-machine-reset operation.

  A *valid* access-list entry is one that has been allocated by an Add ALE operation of the virtual machine and designates an address space that the virtual machine has permission to access.

  A *revoked* access-list entry is one that has been allocated by an Add ALE operation of the virtual machine, but that designates an address space no longer permitted access by the virtual machine. This condition can happen when the owner of the address space explicitly isolates the address space from other users or destroys the address space. The revoked state is different from the unused state in that an ALE can enter the revoked state without any action by the owner of the access list. It is not necessarily a programming error for an ALE to be in the revoked state, and having a separate state pro-

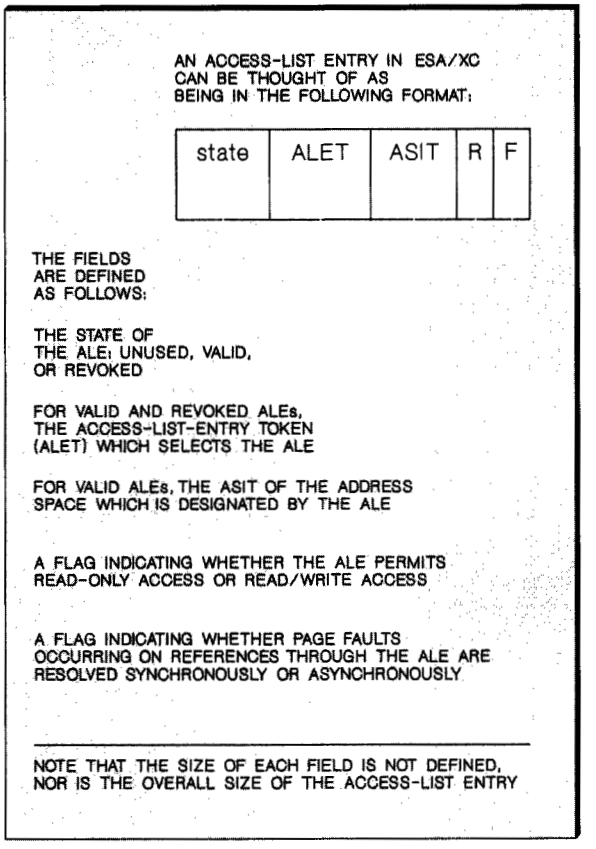
vides additional information to the owner of the access list.

- *A read-only or read-write indication*—When the program adds an address space to its access list, it stipulates whether accesses using the resulting ALET must be read-only accesses or whether they can be read-write accesses. The program cannot, of course, obtain read-write access when the owner has granted it read-only permission, but it can stipulate read-only access to an address space to which it has been granted read-write permission. The program may even add the same address space to its access list several times, some entries allowing read-write access, and others merely read-only access.
- *Synchronous or asynchronous fault resolution*—It was mentioned in the introduction that to make efficient use of storage, a supervisor or host might temporarily "page out" data from an address space to auxiliary storage. ESA/XC leaves the storage management of ESA/XC address spaces up to CP, the host. In order to make efficient use of its own storage, CP sometimes pages out parts of an address space. When an application program references a paged-out portion of an address space, CP receives an indication of this action, called a "page fault." CP will then read the data back into storage and rerun the virtual machine. Except for the slight time delay involved, the virtual machine is unaware of this activity and proceeds normally.

  For reasons described later, it is sometimes undesirable to delay the execution of the virtual machine. CP thus provides the program with the ability to specify, when adding an entry to its access list, whether page faults should be resolved synchronously or asynchronously. A program that specifies asynchronous page-fault resolution is not delayed by page faults but must contain additional logic to complete the "handshaking" necessary to make use of the function.

Figure 2 shows the format of an ESA/XC access-list entry graphically.

Figure 2 A graphic representation of an access-list entry



Using the access-register mode. Application programs using ESA/XC run in one of two address-space-control modes: primary-space and access-register (AR) mode. The primary-space mode is the initial address-space-control mode and the "normal" mode for most CMS applications. The access-register mode is the address-space-control mode in which an application program will run to access data in nonprimary address spaces.

The address-space-control mode of a virtual CPU is determined by bits in the program status word (PSW). Such PSW control bits are normally manipulated only by CMS services, but ESA/XC includes a nonprivileged instruction called SET ADDRESS SPACE CONTROL (SAC) to be used by an application program to switch between the primary-space mode and the access-register mode.[5]

Once in the access-register mode, the application program makes use of 16 registers called access registers to specify to the machine what address spaces are to be accessed by what references. Nonprivileged instructions are provided to allow the program to manipulate access registers. These instructions are the same as those defined

Table 1 Interfaces to basic ESA/XC functions

Function	CP Macro	CMS CSL Routine
Create Space	ADRSPACE CREATE	DMSSPCC
Permit Space	ADRSPACE PERMIT	DMSSPCP
Isolate Space	ADRSPACE ISOLATE	DMSSPCI
Destroy Space	ADRSPACE DESTROY	DMSSPCD
Query Space	ADRSPACE QUERY	DMSSPCQ
Add ALE	ALSERV ADD	DMSSPLA
Remove ALE	ALSERV REMOVE	DMSSPLR

in ESA/390. An access register should either contain zeros or an ALET which was given to the application program by CP when the application program added an entry designating an address space to its access list. Each access register is associated with the general register of the same number. When that general register is used as a "base register" during an instruction, the corresponding access register is examined by the machine to determine what address space is being referenced. If the access register contains zeros, the reference is to the primary address space. If the access register contains an ALET selecting a valid access-list entry, the reference is to the address space designated by that entry. If the selected ALE is in the unused or revoked state, a program interruption will result.

The process in the ESA/XC architecture of converting an access-register number into an address-space identification is called access-register translation, or ART. Figure 3 shows the ART process graphically.

In this example, a LOAD instruction is executed in AR mode. The instruction addresses a storage operand using general register 12 as a base register and general register 3 as an index register. The effective address is calculated just as in ESA/390 and 370-XA by summing the contents of the base register, the contents of the index register, and the displacement, which in this example is 100. This address is either a 31-bit address or a 24-bit address, depending on the current addressing mode of the virtual CPU, indicated by bit 32 of the PSW. The effective address is the location in the address space of the data but does not indicate what address space is to be used. Since general register 12 was used as the base register, access register 12 is the register that determines the address space in which the operand resides. The access register contains an ALET. The diagram shows that if the ALET is zero, the primary address space is the affected one. If the ALET is nonzero, it selects an access-list entry in the access list of the virtual machine. The selected access-list entry, if valid, contains an ASIT, which selects the affected address space.

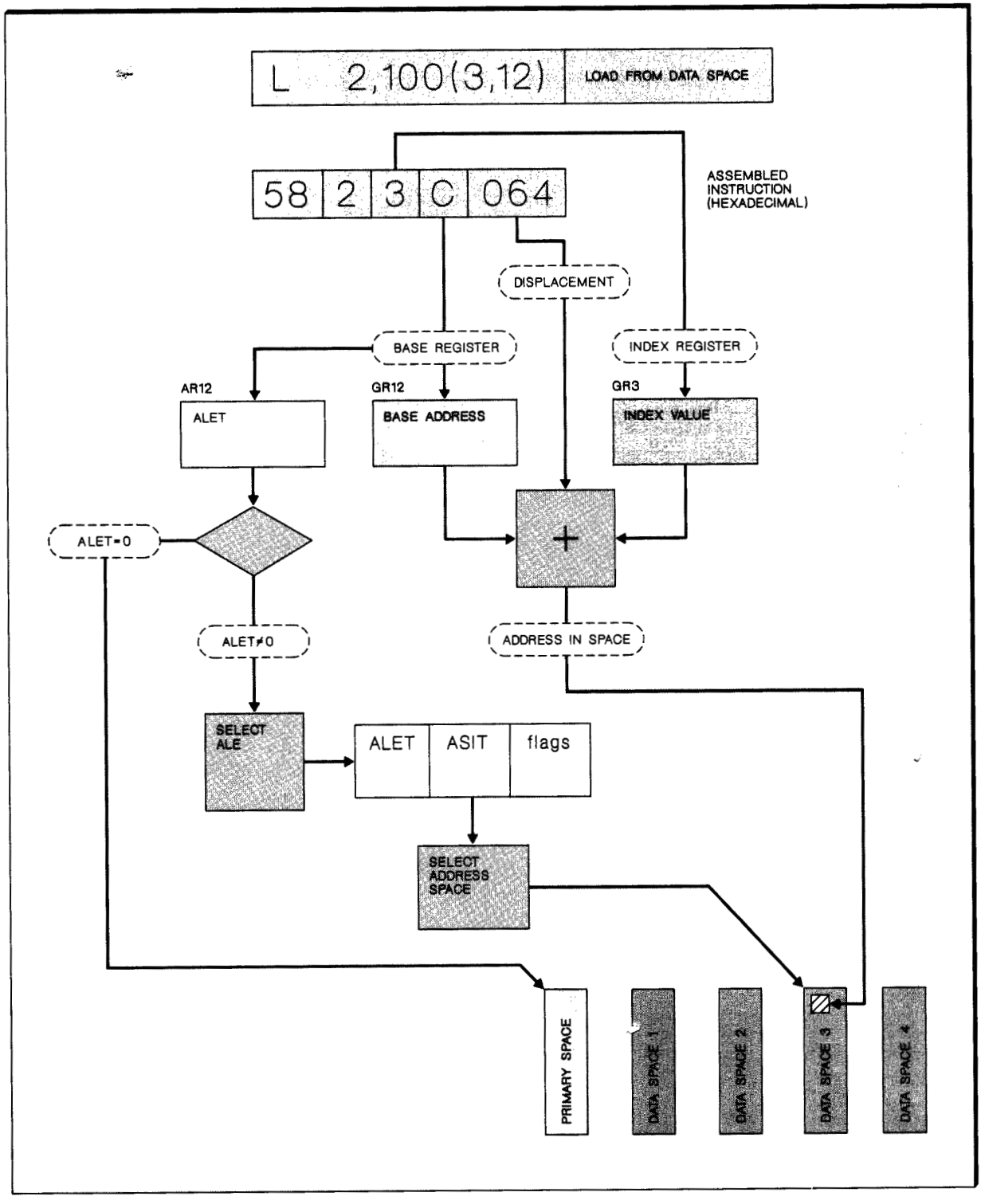
Basic ESA/XC services

The ESA/XC virtual-machine architecture is complemented by a collection of VM/ESA services for creating and managing data spaces and for managing access to address spaces.[6] This section introduces the basic set of VM/ESA services involved in using the ESA/XC advanced-addressing capabilities, and through an extended example, puts the use of these services into context. The descriptions of the services provided in this section are not meant to be exhaustive; for many of the functions, there are other parameters and options besides the ones described here.[7,8] Rather, this section attempts to highlight the major operations that an application uses in order to exploit the advanced-addressing functions.

These VM/ESA services are available as both CP and CMS interfaces, as shown in Table 1. Since the structures related to ESA/XC are managed by CP, the CP interfaces define the basic set of available operations. The CP interfaces are available as macros. The CMS interfaces serve as "cover" functions on top of the CP interfaces to make CMS aware of application-program use of the advanced-addressing capabilities. This allows CMS to perform resource cleanup and recovery functions consistent with CMS philosophy. For example, CMS can be directed to delete implicitly certain data spaces after events of which CP is unaware, such as at end of command or after abnormal program termination. The CMS interfaces are provided as Callable Services Library (CSL) routines. Although both the CP and CMS interfaces are fully supported as general programming interfaces, application programs running on CMS are encouraged to use the CMS interfaces to receive the benefit of CMS-managed recovery and cleanup.

Creating a data space. To begin the description of the ESA/XC services, suppose that a program running in an ESA/XC virtual machine named JMG wishes to create a data space. It will subsequently share this data space with other virtual machines on the same VM/ESA system.

Figure 3 ESA/XC access-register translation: ESA/XC ART converts an access-register number into an address-space identification to determine the address space to be used for the operand access



To create this new data space, JMG invokes the Create Space function. The Create Space request specifies the size of the new data space in 4K-byte pages and an address-space name to be assigned to the new data space. In this example assume that the data space is to be named G53DATA and will be 8192 pages in size. In response to the Create Space request, VM/ESA builds the software and hardware structures associated with the data space and then returns to the caller the address-space-identification token (ASIT) associated with the newly created data space. For this example, assume that an ASIT of X'1234' is assigned to G53DATA.[9]

As described earlier in this paper, the ASIT is the "handle" by which VM/ESA references an address space. It is required as an input to identify a particular address space for subsequent function requests, for example, to establish addressability to the space or to delete it. Therefore, the application must preserve the ASIT returned by the Create Space function.

Because the control structures and resident data pages for a data space consume real system storage, the installation has been given control over the ability of a virtual machine to create data spaces. The user directory entry for a virtual machine specifies upper limits on the number of data spaces and the total amount of data-space storage that the virtual machine is authorized to own simultaneously. If a Create Space request would exceed these limits, it is rejected.

Once a data space has been created, it persists until its owner explicitly deletes it using the Destroy Space function (described later), or until a virtual-machine reset occurs on the owning virtual machine. If the Create Space function was invoked via the CMS CSL interface, the data space may also be deleted implicitly by CMS at end of command or as part of cleanup for abnormal program termination.

Granting access to address spaces. Assume now that JMG wishes to share data space G53DATA with a program running in virtual machine JPH and that JPH is to have read-only access to the data space. JMG will require read-write access to the same address space.

JMG indicates that JPH is authorized to access G53DATA by invoking the Permit Space function. The Permit Space request includes the ASIT of the address space for which permission is to be granted (ASIT X'1234' for data space G53DATA in the example), the identity of the target virtual machine (userid JPH), and the type of permission: read-only or read-write. As a result of the Permit Space request, VM/ESA records that virtual machine JPH has been authorized for read-only access to the address space. Note that Permit Space authorizes a virtual machine to obtain access to an address space, but it does not actually establish that access. The target virtual machine must subsequently perform the Add ALE function (described in the next subsection) to establish access.

Figure 4 illustrates the situation after data space G53DATA has been created and virtual machine JPH has been permitted to access it.

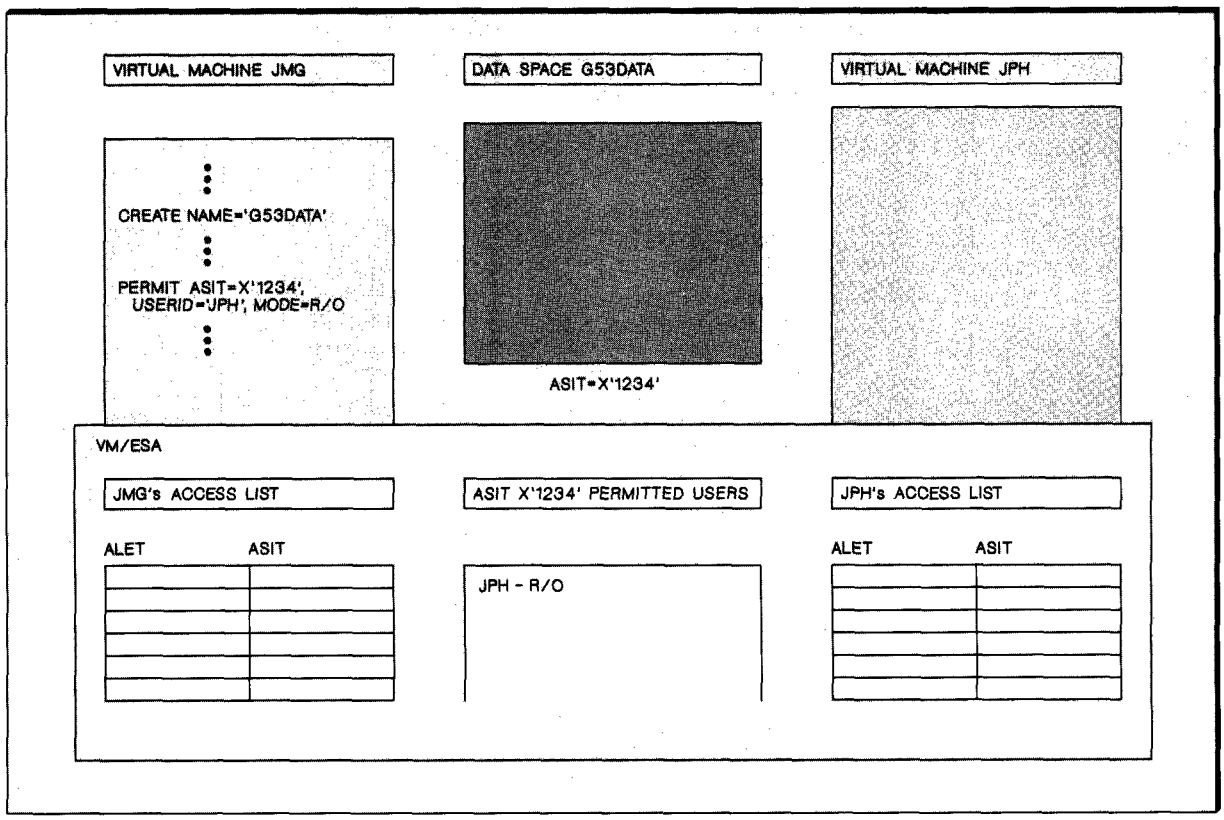
Since JPH will need to know the ASIT assigned to G53DATA in order to establish addressability to the space, JMG would normally inform JPH of the ASIT after it granted permission. Standard VM/ESA communication facilities such as advanced program-to-program communications/virtual machine (APPC/VM) or IUCV can be used for this purpose. Often, these APPC/VM or IUCV communication paths already exist between the virtual machines for other reasons, as for example, for normal communication between a client and a server, so new paths do not have to be established.

The ability to specify the read-only or read-write permission separately for each permitted virtual machine is one of the advantages of using shared address spaces over shared segments to share data. Shared segments can be defined as read-only or read-write, but this attribute is the same for all virtual machines that share the segment. Address spaces shared via ESA/XC facilities can have any combination of read-only and read-write users. This ability allows structures, for example, in which one trusted virtual machine is allowed to write data into an address space that can be only read by other less-trusted virtual machines.

JMG does not have to use Permit Space to give itself access to G53DATA since the owner of an address space always has read-write permission to the space. In fact, it would be an error for JMG to try to give itself permission.

The first Permit Space request for a particular address space transforms that space from a pri-

Figure 4 Creating an address space and authorizing access to it



vate address space into a shareable address space. The address space remains shareable until it is isolated or destroyed. Because this transformation allows other virtual machines to access the address space, it changes treatment by the VM/ESA paging subsystem of the address space as well. When an address space is private, only the owning virtual machine can reference it, so CP can attribute to the owner all reference activity to the address space. The VM/ESA paging subsystem therefore manages a private address space in conjunction with all other private storage owned by the same user, applying page-selection criteria to all of these address spaces uniformly, depending on the reference pattern and activity of that user.[10] In contrast, when an address space is shareable, references to it may be made by any of the permitted virtual machines, so it is no longer possible to attribute activity to any particular user. As a result, the VM/ESA paging subsystem disassociates a shareable address space from the owning user and, instead, treats the address space as a form of system-wide shared storage. This treatment results in more global consideration of the address space for paging purposes. Because use of the Permit Space function results in this change in paging treatment, a virtual machine must be authorized in the user directory to employ it.[11]

The permission to access an address space that is conferred by Permit Space persists until it is revoked by the owner (via the Isolate Space function described later), until the address space is deleted, or until a virtual-machine reset occurs on either the owning or the permitted virtual machine.

Finally, it is worth noting that although the example focuses on sharing a data space, primary address spaces can be shared as well. Virtual machine JMG could have given JPH access to its primary address space in essentially the same man-

ner as just described for data space G53DATA. The only difference is that since the primary space is not created via the Create Space function, the virtual machine does not have the ASIT for the primary space available. However, it can use the Query Space function to obtain this ASIT.

Establishing addressability. Now that JMG has created data space G53DATA and authorized JPH to access it, the next step in the process is to establish addressability for the data space. Since both JMG and JPH wish to access the data space, this operation will be performed by both virtual machines.

The Add ALE function is used to obtain access to an address space. The Add ALE request specifies the ASIT of the address space and the type of access (read-only or read-write) desired. In the example, both JMG and JPH invoke the Add ALE function, specifying the ASIT for G53DATA (X'1234'), since they both want to access this space. JPH requests read-only access; it would have been an error if it had requested read-write access since it only has permission for read-only access. JMG is the owner of the address space and is thus free to request either type of access. JMG might request read-only access if that is all that is required, perhaps to eliminate the possibility of an erroneous store into the address space.

The result of the Add ALE request is that an unused entry in the access list associated with the virtual machine is placed in the valid state and set to designate the requested address space. The ALET corresponding to the selected access-list entry is returned to the caller. As previously described, the collection of valid entries in the access list constitutes the set of address spaces that a virtual machine can reference using the advanced addressing features provided by ESA/XC.

The ALET returned by Add ALE is the handle used by the hardware to designate the address space containing an instruction operand. The ALET designates a particular entry in the access list associated with the virtual machine; the access-list entry in turn designates the address space. Since access lists are managed independently for each virtual machine, the ALET assigned for the accesses of one virtual machine to an address space is usually different than the ALET assigned for the accesses of another virtual machine to the same space. For example, ALET X'0022' may be returned for use by JMG, whereas ALET X'0008' may be returned for JPH.[12]

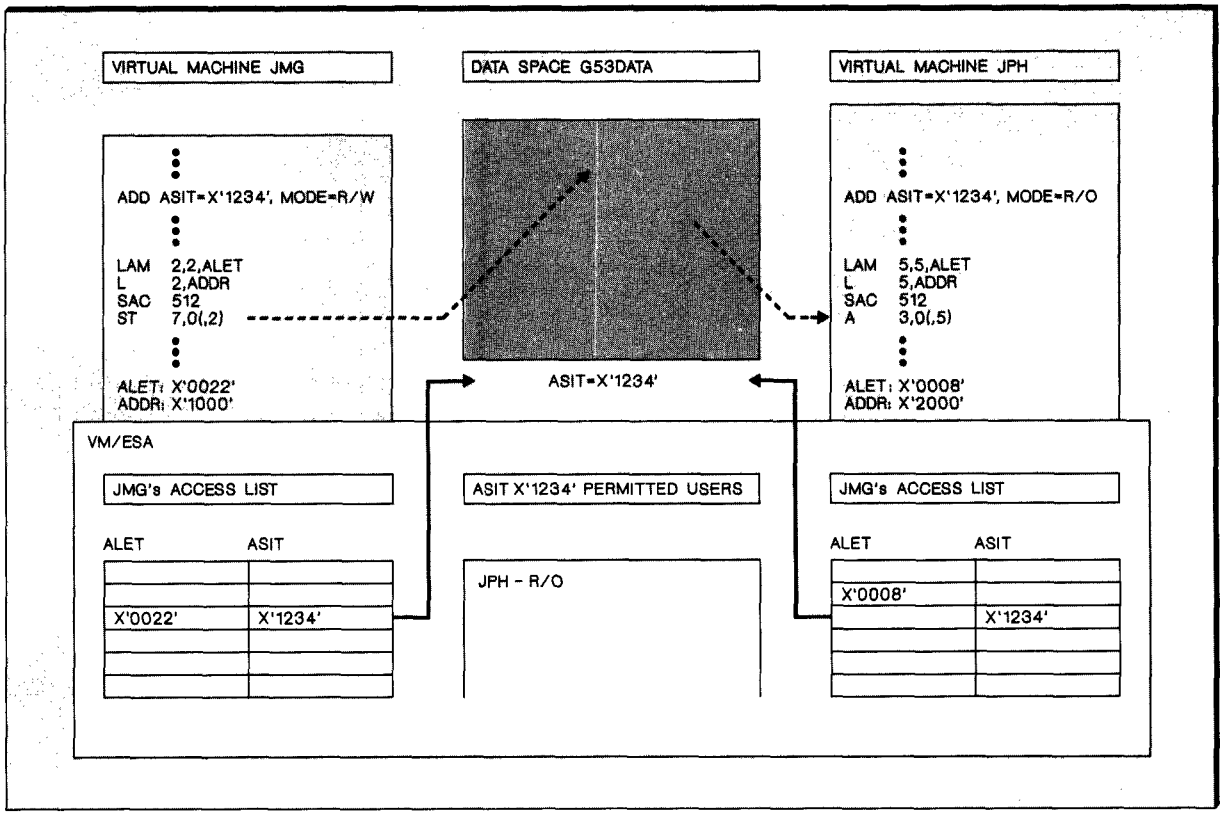
This example points out an important difference between ASITs and ALETs. ASITs are managed so that they have systemwide scope. Both JMG and JPH used the same ASIT to identify data space G53DATA to the system. In contrast, an ALET is translated through the access list associated with a virtual machine, so the address space selected by a particular ALET value depends on the contents of that list. The ALET therefore has only local scope. If JMG attempts to use the ALET assigned to JPH for references to G53DATA, it is very likely that either the reference made by JMG will not succeed (result in a program exception) or will occur in the wrong address space. Programs that run in different virtual machines and share address spaces must be aware of these differences between ASITs and ALETs and use the tokens appropriately.

Referencing address spaces. Thus far, all of the setup operations required to use the ESA/XC advanced-addressing capabilities have been described: creating address spaces, permitting other virtual machines to access address spaces, and establishing addressability to address spaces.

To fetch data from or store data in an address space other than the primary address space, a program places the appropriate ALETs into the access registers that will be used for storage operands and enters the access-register (AR) mode. While in this mode, each storage operand is designated in a *space.address* form via an access-and-general register pair: the ALET in the access register indicates the address space containing the operand, and the address in the general register (possibly modified by a displacement or index specified in the instruction) indicates the location of the operand within the address space. The access-and-general register pair used to designate a storage operand corresponds to the base-register-number field in the instruction for the storage operand. For instructions such as MOVE (MVC) that have multiple storage operands, each operand can be designated by a different access-and-general register pair, and therefore the operands can reside in different address spaces. This process has been described in more detail earlier in this paper.

As we continue with the example, Figure 5 shows both virtual machines JMG and JPH referencing

Figure 5 Referencing data in G53DATA



data in data space G53DATA. The LOAD ACCESS MULTIPLE (LAM) instruction is used to load ALETs from storage into access registers, and the SET ADDRESS SPACE CONTROL (SAC) instruction is used to switch between the primary-space and AR modes. Once in the AR mode, the storage operands of normal CPU instructions such as STORE or ADD can reside in address spaces other than the primary space.

Given that the proper setup operations have been performed, the process of entering AR mode and referencing data in that mode is accomplished without intervention by CP or CMS. The addressing operations are performed entirely by the machine. If proper setup has not been performed, the error condition is reported to the virtual machine as a program interruption. CMS in turn converts the program interruption into a program abnormal end (abend) and drives abend-handling exits to invoke application cleanup.

While executing in AR mode, some ART-related program interruptions may be recognized as a result of improper setup. For example, an exception is recognized if a program uses an ALET that designates an unused or nonexistent access-list entry. In addition, a program interruption may be recognized if a program attempts to reference an address space for which its access permission has been revoked. This condition is not due to improper setup, but rather results if the owner of an address space isolates or destroys the address space "out from under" the accessing virtual machine.

In most cases, it is necessary to make references to the primary address space while in AR mode, for example, to move data from the primary space into or out of a data space. A valid access-list entry designating the primary space is not required in order to do this (although there is nothing preventing such an access-list entry from be-

ing established). Rather, the special ALET value of binary zero has been reserved to designate the primary space for use in such cases. This ALET value is translated without using the access list.

Dropping addressability. When access to a particular address space is no longer needed, a virtual machine can drop addressability to it by using the Remove ALE function. This function places a valid (or revoked) access-list entry in the unused state so that the entry can be used again later. The access-list entry to be removed is indicated by specifying the ALET associated with it.

The Remove ALE function affects only current addressability to the address space; it does not change permission obtained by a virtual machine to the address space. So, for example, as long as JMG does not revoke the access of JPH to data space G53DATA, JPH is free to use the Add ALE and Remove ALE functions to establish and drop addressability to G53DATA as often as necessary.

Generally, no penalty is associated with maintaining addressability to an address space for longer than required. Thus for most applications, the Remove ALE function is needed only during application termination. However, an extremely complex application may require access to more address spaces than it has access-list entries. Such an application must actively manage the contents of the access list associated with the virtual machine, temporarily removing addressability to an address space to make room in the access list for addressability to some other space. The Remove ALE function makes such management possible.

As part of virtual-machine reset, the Remove ALE function is performed implicitly for each entry in the access list associated with the virtual machine. As a result, at the completion of virtual-machine reset all of the access-list entries are in the unused state.

Isolating address spaces. The owner of a shared address space can return it to the private state through the use of the Isolate Space function. This function may be useful in preparing to make widespread changes to an address space, or indicating that a later version of data exists in some other address space.

Isolate Space is a global revoke operation for an address space. It rescinds all permissions to the address space previously granted via the Permit Space function and cancels any current addressability that other virtual machines may have to the address space. Any addressability that the owning virtual machine has to the address space being isolated remains unchanged. The Isolate Space function performs these operations in such a way as to make the important guarantee that when the Isolate Space function completes its operation, only the owning virtual machine is able to access the address space that was isolated.[13]

Current addressability of other virtual machines is canceled by changing applicable entries in their access lists from the valid to the revoked state and then terminating any in-progress storage accesses to the address space. In the absence of other communication between the owner and the sharing virtual machine, the sharing virtual machine will find out that its access was revoked via a program interruption on its next attempt to reference the address space.

Figure 6 continues the example, showing the state of data space G53DATA and the access lists for virtual machines JMG and JPH after JMG has isolated G53DATA.

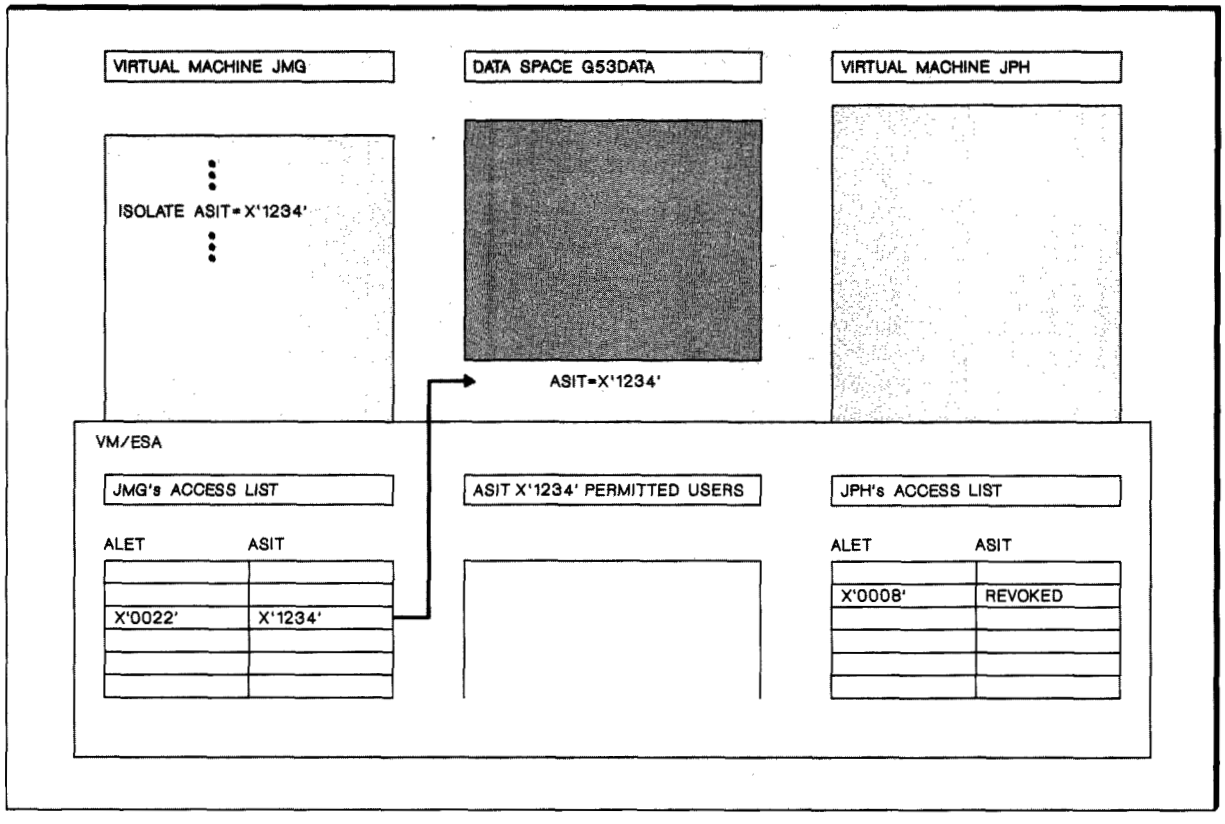
Because the Isolate Space function returns an address space to a state in which only the owner can access it, all reference activity to the address space is once again attributable solely to the owner (as was the case before the first Permit Space request for the address space). Therefore, after an Isolate Space operation, the VM/ESA paging subsystem resumes private-storage treatment for the address space.

If not performed earlier, an Isolate Space operation is performed implicitly on each address space owned by a virtual machine as part of virtual-machine reset.

Destroying data spaces. When a data space is no longer needed, the owner of the data space can destroy it using the Destroy Space function. Since a data space always consumes some amount of real system storage, it is good practice to destroy data spaces when finished with them.

The data space to be destroyed is specified by an ASIT. The Destroy Space function discards the current contents of the data space and frees all control structures used to represent the data

Figure 6 Isolating an address space



space. If the data space being destroyed is a shareable address space, an implicit Isolate Space operation is performed as part of the Destroy Space operation to terminate the access of other virtual machines to the data space.

Since the Destroy Space function terminates not only the accesses of other virtual machines to the data space, but the owner's accesses as well, any entries in the owner's access list that designate the data space being destroyed are set to the revoked state. If the CMS CSL interface is being used, CMS implicitly performs the Remove ALE requests to return those entries to the unused state, thus making those entries available for reuse.

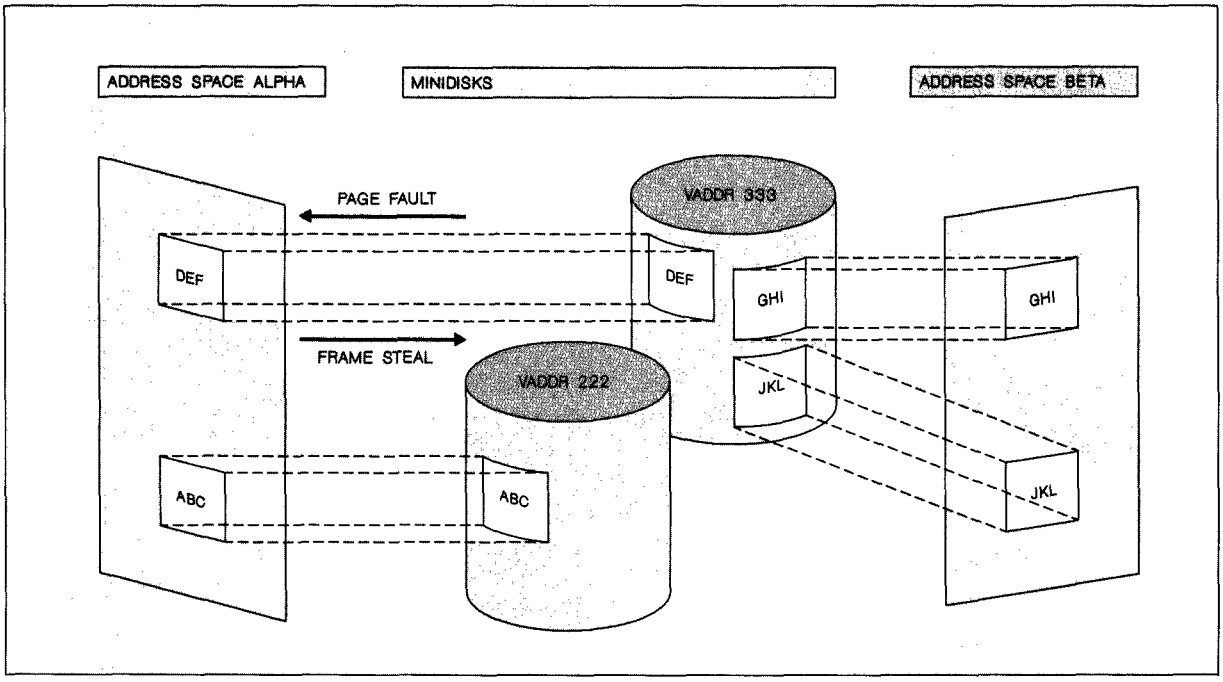
If not performed earlier, an implicit Destroy Space operation is performed on every data space owned by a virtual machine as part of virtual-machine reset.

Advanced ESA/XC services

In addition to the basic ESA/XC system functions described in the preceding section, VM/ESA also provides a collection of advanced functions for more sophisticated exploitation of ESA/XC capabilities. These additional functions include services for performing data-in-storage mapping between minidisks and address spaces, handshaking to allow asynchronous processing of page faults, and a service to optimize system handling of application data. These functions are described briefly in the following subsections.

Mapping services. For some applications, data spaces might be useful as a fast way of referencing data that are permanently stored on DASD. Temporarily placing the data in data spaces and referencing the data in those data spaces could improve application performance, particularly if the same data are used repeatedly. As long as the data-space pages remained resident in processor

Figure 7 Mapping minidisk data into address spaces



storage, access to the data would occur very quickly.

One way of structuring such an application would be to load a data space or set of data spaces with data from DASD at the beginning of the execution of the application, and to store the changed data from the data space back in the DASD locations at the conclusion of the application. Unfortunately, in many cases such an approach would be inefficient, especially if the application usually only processed a small, randomly accessed portion of its data, or if it did not keep track of the specific data items that were changed, but rather rewrote all of the data at the conclusion of the application. The I/O operations wasted in loading and storing the portion of the data that was never accessed or changed could more than offset the benefit received from quick access to the portion that was needed. The processor storage wasted for unneeded data could cause a sharp increase in the paging activity performed by the system, resulting in a system-wide performance degradation.

The mapping services provide a way to use data spaces as a temporary repository for DASD data while minimizing unnecessary I/O operations or storage overhead. These services are similar in nature to the MVS/ESA data-in-virtual services described by Rubsam.[14] Through the mapping services, a program can establish an association between a collection of minidisk blocks (on one or more minidisks) and a collection of address-space pages (in one or more address spaces). As shown in Figure 7, when a mapping exists, an image of the data that resides on the mapped minidisk blocks appears in the associated address space pages without the need to perform application-requested I/O operations to load the data. Instead, the I/O operations are performed implicitly by the VM/ESA system as references are made to the mapped address spaces. In addition, the mapping services allow a program to save only the changed mapped data back on the minidisks without keeping track of the specific pages that were changed (the CP paging subsystem tracks the changes).

The mapping services are provided by the CP MAPMDISK macro as shown in Table 2; there is currently no CMS interface for the mapping services. However, CMS applications can invoke the mapping services via the CP interfaces.

The Identify Pool function is the first mapping function used by an application. It identifies the collection of minidisks in the I/O configuration of the virtual machine that will participate in mappings. This collection is known as the minidisk pool. For each block within the minidisk pool, the Identify Pool request specifies a unique pool-relative block number to be assigned to the block. This number is used by the mapping services to establish the association between address-space pages and blocks within the minidisk pool.

Once the minidisk pool has been identified, an application uses the Define Mapping function to establish mappings between address-space pages and minidisk blocks. The Define Mapping request specifies a range of address-space pages to be mapped and also specifies, for each page in the range, the pool-relative block number identifying the minidisk block to be associated with the page. Immediately upon completion of the Define Mapping operation, the data that are contained on the minidisk blocks are available in the newly mapped pages. That is, a reference to a mapped page via a processor instruction such as LOAD will "see" the data that are on the associated minidisk block. The mapped data can be changed by simply changing the appropriate address-space locations.

Again, even though it is said that the data are available immediately, actual movement of data from a minidisk block into the address space is deferred until the first reference is made to a page. The Define Mapping operation sets all of the newly mapped pages such that any reference to them causes a page fault to occur. When the mapped page is referenced and the page fault is generated, the data are read from the minidisk block by the VM/ESA paging subsystem as part of the process of resolving the page fault.

After a mapped page is in storage, the VM/ESA paging subsystem may decide to "steal" the storage frame associated with the mapped page. If the frame is stolen and has been changed (determined by inspecting the hardware change bit for the page), the paging subsystem writes the changed page back on the associated minidisk block, thus preserving the changes. If a virtual machine makes another reference to the mapped page, it will be reread from the minidisk block.

Table 2 Interfaces to mapping services

Function	CP Macro
Identify Pool	MAPMDISK IDENTIFY
Define Mapping	MAPMDISK DEFINE
Save	MAPMDISK SAVE
Remove Mapping	MAPMDISK REMOVE

Although the VM/ESA paging subsystem may sometimes cause changed, mapped data to be saved on the minidisks, from the point of view of a virtual machine it is unpredictable if (and when) the paging subsystem may perform this action, and so the application cannot depend upon it. Instead, the application can use the Save function to guarantee that changed, mapped data are stored back on the minidisks. The Save request initiates an asynchronous operation that writes to the associated minidisk blocks any of the pages in a list of mapped pages that have not been written out since they were last changed. Pages for which the minidisk copy is up-to-date are not rewritten. When completion of the save operation is indicated via an interruption to the requestor, the application is guaranteed that the minidisk blocks contain a current copy of the mapped data.

Finally, when an application is finished referencing the mapped data, it can eliminate the mapping association by using the Remove Mapping function. This function restores the address-space pages to "unmapped" status so that they can be used as normal virtual-machine storage.

The Structured Query Language/Data System (SQL/DS) database product and the CMS Shared File System both use VM/ESA mapping services in conjunction with the exploitation of data spaces by those components.

Asynchronous page-fault handling. As was mentioned earlier, the address spaces that are available to ESA/XC virtual machines are managed by CP. In an effort to make the most efficient use of the real machine storage, CP makes decisions about what portions of virtual-machine storage should and should not be resident in real storage. If a virtual machine references a portion of an address space that is not resident in real storage, a page-fault indication is presented to CP. When normal, synchronous, page-fault resolution is in effect, CP responds to the page-fault indication by

suspending execution of the virtual machine, performing the paging operation to make the required pages resident, and resuming execution of the virtual machine when the paging operation is completed. The page fault and its subsequent resolution by CP are transparent to the virtual machine except for the time delay during which the virtual machine was suspended.

For most applications, the time delay incurred in synchronous page-fault resolution is not significant. However, for certain applications, such as servers, the delays caused by synchronous resolution can be a problem. These servers perform functions (for example, database management) on behalf of many users of a VM/ESA system. Such a server is usually structured to use a form of multitasking to maximize server throughput. Each incoming user request is assigned by the server to a separate task. When progress on one task is delayed, for example, to wait for an I/O operation to complete, the server switches to work on another task for which progress can be made. In doing so, the server makes productive use of the relatively long time between I/O initiation and I/O completion instead of wasting the time by simply waiting.

Unfortunately, this technique cannot be used to avoid delays caused by page faults when the faults are being resolved synchronously. Since CP suspends the server during the fault-resolution process, the server does not have an opportunity to run another task while the page fault is being resolved for the faulting task. That is, all of the tasks of the server are delayed while CP is busy resolving a page fault incurred by just one of them. As a result, server throughput is degraded.

In contrast, asynchronous page-fault handling allows a server to productively run other tasks while a page fault is being resolved for an AR-mode reference made by one task. When a page fault occurs during an AR-mode reference and asynchronous resolution is enabled for that reference, CP initiates the paging operation to make the required page resident. CP then immediately resumes execution of the virtual machine, presenting a page-fault-initiation interruption to the virtual machine as a signal that a page fault has occurred. The multitasking server in the virtual machine normally responds to the page-fault-initiation signal by suspending its current task (the one that just encountered the page fault) and selecting some other task to run. The new task runs in parallel with the CP paging operation to resolve the page fault of the original task.

When the CP paging operation is completed and the required pages are resident, CP presents a second signal, a page-fault-completion interruption, to the virtual machine. This interruption indicates to the server that the faulting task does not have to be delayed any longer since the pages it requires are now available in storage.

Multiple instances of asynchronous page-fault resolution may be outstanding at the same time. For example, task B, which was run when task A encountered a page fault, may itself encounter a page fault before the page-fault-completion signal is received for task A. The page fault on task B initiates another instance of asynchronous page-fault resolution and will result in another pair of initiation and completion signals, this time for task B. Because multiple page-fault-completion signals can be outstanding at the same time, a method is needed to identify for what task a particular page-fault-completion signal is intended.

Associating the initiation and completion signals with the intended task is accomplished by providing a task-identification token as part of the signals.[15] Before enabling asynchronous page-fault handling, a server uses the CP PFAULT macro to specify the location of a word in storage that it maintains as the task-id token of the current (running) task. Typically, this word is set by the server to be the address of the "task control block" for the current task. When a page fault occurs, CP obtains the token for the current task from this server-specified location and supplies this token in the page-fault initiation and completion signals for the page fault just encountered. This action allows the server to process the page-fault-completion signal without performing any "task control block" searching. Such searching would be required if a CP-generated, rather than server-supplied, token were used in the signals.

Asynchronous page-fault handling is enabled or disabled independently for each entry in the access list associated with a virtual machine. When an ALE is established via the Add ALE service, a parameter on that request indicates the type of page-fault treatment desired for references made

via the ALE. The default is synchronous page-fault handling. This flexibility allows a server to use asynchronous page-fault handling for some types of references, while avoiding it for other types that cannot tolerate the "loss of control" implied by task suspension and resumption. Asynchronous page-fault handling is not available for references that do not involve use of an ALE. That is, references made while in the primary-space mode or references made with an ALET of binary zero while in the access-register mode are always handled synchronously.

Finally, it is worth noting that asynchronous page-fault handling is normally applied only to page faults that require CP to perform an I/O operation to resolve them. If a page fault can be handled without an I/O operation, for example, because the required data are available in expanded storage, the fault is almost always handled in the synchronous manner. The time to process such page faults is so short that the extra overhead involved in asynchronous handling is not beneficial.

Reference pattern notification. Asynchronous page-fault handling reduces the system-wide impact of page faults by allowing a multitasking application to overlap other processing with the resolution of a page fault. But asynchronous page-fault handling is not the complete solution: The task that encountered the page fault is still delayed, and single-task applications cannot easily exploit the asynchronous page-fault handling capability. An even better approach is to avoid page faults completely by making virtual-machine storage resident (just) before it is needed.

To this end, CP provides the REFPAGE macro to allow an application to give hints to the VM/ESA paging subsystem about the upcoming storage reference patterns of the application. The reference pattern may be regular, as might be encountered during the processing of large arrays. Alternatively, the reference pattern may be irregular, appearing as a collection of unrelated references with the overriding pattern being visible only at a higher level. Such a reference pattern may be encountered during the indexed scan of data of a database server; the individual references appear unrelated, but are in fact predictable based on the contents of the index.

As a result of the REFPAGE hints, the VM/ESA paging subsystem alters its normal formation of "paging blocks" based on past recency of reference to instead form blocks that are related to the stated future reference pattern of the application. When a page fault occurs on one page in a paging block, the paging subsystem also makes resident some or all of the other pages within the block (depending on system storage availability). To the extent that the paging blocks mirror the reference pattern, this block-oriented paging eliminates page faults on subsequent references to other pages.

Specifying reference hints that do not closely match the actual reference pattern can result in unneeded pages being made resident.

As is the case for tuning functions in general, incorrect application of the REFPAGE macro can degrade rather than improve performance. Specifying reference hints that do not closely match the actual reference pattern can result in unneeded pages being made resident, possibly displacing other pages that were useful; this could result in increased paging. Therefore, the REFPAGE macro should be judiciously applied.

Concluding remarks

IBM's ESA/390 architecture was developed and refined after careful consideration of the requirements of sophisticated operating systems managing real-machine resources. It is IBM's strategic architecture for such operating systems. IBM's new ESA/XC architecture is a derivative of ESA/390 and makes the advanced address-space facilities of ESA/390 available to the CMS application program. ESA/XC has been shaped specifically to satisfy CMS requirements and is thus IBM's strategic architecture for supporting the CMS environment.

ESA/XC provides an environment where programs can create additional storage in the form of data

spaces and share those data spaces or the primary address space with other virtual machines. The virtual machine does this without being encumbered by the chore of maintaining DAT and ART tables, or by paging to efficiently use storage; these are taken care of by CP. Exploiting additional address spaces provides better reliability through data separation, enriched function through high-speed and flexible data sharing, and increased capacity from the ability to concurrently address more than 2G of storage. Mapping data on DASD into an address space likewise provides extended capabilities suitable for a variety of purposes. ESA/XC is the next step in the natural progression of virtual-machine architectures. It has the same advantages over the System/370 architecture that 370-XA has, but the additional ESA/XC-only services make ESA/XC superior to 370-XA for CMS applications. The transition from 370-XA to ESA/XC is much smoother than was the transition from System/370 to 370-XA.

ESA/XC is provided by collaboration between VM/ESA and the machine. Many such synergetic relationships are possible and could conceivably provide further useful function for the CMS programmer. ESA/XC is an extendable base for such improvements.

Cited references and notes

1. K. E. Plambeck, "Concepts of Enterprise Systems Architecture/370," *IBM Systems Journal* **28**, No. 1, 39–61 (1989).
2. *IBM Enterprise Systems Architecture/390 Principles of Operation*, SA22–7201, IBM Corporation; available through IBM branch offices.
3. D. L. Osisek, K. M. Jackson, and P. H. Gum, "ESA/390 Interpretive-Execution Architecture, Foundation for VM/ESA," *IBM Systems Journal* **30**, No. 1, 34–51 (1991, this issue).
4. A 64-bit token provides, of course, a finite number of unique values (namely 2^{64}). If a system were to run forever, therefore, it would be impossible to guarantee uniqueness indefinitely. The algorithm chosen by VM/ESA does not generate all 2^{64} possible ASIT values but does provide a sufficient range to allow this guarantee to be made in practice.
5. A detailed description of SAC and other instructions in ESA/XC may be found in the document *VM/ESA Enterprise Systems Architecture/Extended Configuration Principles of Operation*, SC24-5594, IBM Corporation; available through IBM branch offices.
6. The term *data space* is used specifically to refer to an address space created at the request of a virtual machine via the Create Space function. Many of the operations described in this section can also be performed against the primary space of a virtual machine. In such cases, the more general term *address space* is used.
7. *VM/ESA CP Programming Services*, SC24-5520, IBM Corporation; available through IBM branch offices.
8. *VM/ESA CMS Application Development Reference*, SC24-5451, IBM Corporation; available through IBM branch offices.
9. An ASIT is actually eight bytes (16 hexadecimal digits) long. For brevity, a shorter hypothetical ASIT is being used in this example.
10. G. O. Blandy and S. R. Newson, "VM/XA Storage Management," *IBM Systems Journal* **28**, No. 1, 175–191 (1989).
11. When VM/ESA is operating at the United States Department of Defense B1 security level, additional tests must be passed before a Permit Space operation grants permission to another virtual machine. Specifically, mandatory access-control checking is applied to ensure that the permitted virtual machine is at a security level that allows access to the information contained in the address space. If the mandatory access-control checking fails, the Permit Space operation is rejected.
12. An ALET is actually four bytes (eight hexadecimal digits) long. For brevity, shorter hypothetical ALETs are shown here.
13. An obvious omission in the repertoire of ESA/XC services of VM/ESA is an individual Revoke-access function that rescinds permission from particular virtual machines. Although this is not intuitive, an individual Revoke-access function is more difficult to achieve than a global revoke operation, especially in delivering the necessary guarantees about synchronization between the "revoker" and "revokee." Because of these difficulties, and the lack of a strong immediate requirement, the individual Revoke-access function is not included in the initial support of ESA/XC, but is a possible future enhancement.
14. K. G. Rubsam, "MVS Data Services," *IBM Systems Journal* **28**, No. 1, 151–164 (1988).
15. The task-id token is not strictly necessary in the page-fault-initiation signal since that signal is always intended for the current server task. However, for consistency, the initiation and completion signals both provide the token.

Joseph M. Gdaniec *IBM Data Systems Division, P.O. Box 6, Endicott, New York 13760-5553.* Mr. Gdaniec is a senior programmer in the VM System Design department. His current responsibilities include the definition of architecture and VM support for advanced processor capabilities. He was the lead designer for the VM Data Spaces function. Mr. Gdaniec joined IBM in Kingston, New York, in 1982 with an initial assignment in the Scientific and Engineering Processor Products group. He participated in application enabling and benchmarking for the IBM 3090 Vector Facility. In addition, he was coauthor of the VS FORTRAN Program Multitasking Facility, for which he received an IBM Outstanding Technical Achievement Award and a U.S. patent. In 1986, he accepted his current assignment in VM. Mr. Gdaniec received a B.S. in computer science from Indiana University of Pennsylvania in 1982, and an M.S. in computer science from Syracuse University in 1989. He is a member of the Association for Computing Machinery and the IEEE Computer Society.

James P. Hennessy *IBM Data Systems Division, P.O. Box 6, Endicott, New York 13760-5553.* Mr. Hennessy is an advisory programmer in VM System Design at the IBM Endicott Programming Laboratory. He received his B.S. in computer science from Rensselaer Polytechnic Institute in 1982 and joined IBM and VM development the same year. Since then, he has been involved in many enhancements to VM/XA and VM/ESA in the area of real and virtual CPU management and storage management. Most recently, Mr. Hennessy designed and implemented a portion of the VM Data Spaces support available in VM/ESA 1.1.

Reprint Order No. G321-5421.