MVS data services

by K. G. Rubsam

The IBM Enterprise Systems Architecture/370™ vastly increases the potential virtual addressability available to both system and application programs. The I/O model and the application model of permanent data are discussed to illustrate how large virtual addressability can be used to simplify application programs and improve performance. New MVS services that exploit the architecture are described. Also described are data window services, which are callable from high-level languages and provide the capability to manage very large permanent and temporary objects in virtual storage.

Certain operating system features allow application programs to use the large quantities of virtual storage made possible by the Enterprise Systems Architecture/370™. Before describing these features, we review the history of virtual storage in large IBM systems from the time it was included in the System/370 architecture. The single virtual storage (SVS) operating system supports 16M bytes (where $M=10^6$) of virtual addressability for the operating system and the combined total of all the user regions. The Multiple Virtual Storage (MVS) operating system allows each user region and the operating system to have a total of 16M bytes of virtual addressability. MVS/Extended Architecture (MVS/XA™) allows each user region and the operating system to grow to a total of 2G bytes (where $G=10^9$) of virtual addressability.

ESA/370™ is a new architecture that vastly expands the base limits of data addressability. In addition, the architecture allows program exploitation to further extend addressing limits in ways never before possible. ESA raises the architectural limits to allow user's addressability to be expanded in 2G-byte increments so that with register manipulation, up to 16T bytes (where $T=10^{12}$) of data can be addressed. Beyond that, application programs can invoke system services to manipulate tables used by the hardware to extend the addressing capacity by many times more.

New data window services are implemented using concepts built on the architecture to define and access very large temporary and permanent objects. The window services are callable from many of the high-level languages, including FORTRAN and COBOL. Even though the capacity represented by limits on the current hardware configurability will allow applications to increase greatly their present addressable data, those limits are still well below the addressing capability of the architecture and of the window services exploitation of the architecture. This paper provides a historical perspective and describes the new system services that permit application exploitation of the new architecture.

The I/O model vs the application model of data

The physical attributes of storage media have been a very significant factor throughout the history of the data processing industry. In fact, some storage media attributes remain long after the medium itself has become obsolete. It is likely that in almost every data processing installation in the world, one can find direct-access storage device- (DASD) resident datasets with such block sizes as 1600, 3200, and so forth.

These multiples of 80-character logical records are vestiges of the 80-column card that was once one of the mainstays of the data processing industry. The card was more than a machine readable document. It was also the means of storing data when there was a need for editing or updating. This could be done completely electromechanically by creating the new card with a *card punch*, or manually with a machine called a *key punch* in which the card was punched as characters were entered by hand via a typewriter-like keyboard. Card decks were maintained by manually removing the old cards and inserting new ones. The decks were stored in special files with card-size drawers. The card-filled files stood in programmer cubicles and the adjacent halls waiting for the next trip to the machine room, where the cards would be passed through the card reader and returned to the file. Today the cards, the card punches, the card readers, and the file drawers have largely disappeared, but the 80-character records remain.

Access methods were developed to shield applications from the need to build channel programs and to reduce the degree of sensitivity to storage-device characteristics. The access methods not only solve a technical problem for applications, but they also protect them from changes in device geometry that occur when a new DASD is introduced. Naturally, in order to provide this service, the access methods must define some rules. Records can be of variable length or fixed length; maximum record and block sizes are specified. Rules for defining and using buffers are laid out. These rules and the history of data storage lead us to the I/O model of data. If one follows the rules, a degree of device independence can be achieved, which is satisfactory if the application can work with the I/O model of the data. Demonstrably, we are surrounded by examples of applications that work successfully with the I/O model of their data.

There are, however, applications that would be easier to write and extend if their data could be stored in a usable format rather than broken into records and blocks according to rules that are not really relevant to the application programmer's objective.[1] The concept of storing data on DASD in the form exactly as it was created, and subsequently presenting the data in the form the application requires, is called the *application model* of data.

The application model of data means that the application programmer has the same flexibility in manipulating permanent data as when manipulating control structures, pointers, and data in the address space. The application data can be defined with whatever internal structure is needed to meet the needs of the application. There are obvious simplifications in writing an application in this environment.

In addition to contributing to the ease of application development, this form of handling permanent data also allows the operating system to maintain data in processor storage to provide improved performance and to reduce I/O operations. This is possible because the concept of virtual storage has been joined with a system-managed storage hierarchy, using consistent units of storage at a time when large quantities of affordable processor storage are available. These are not all completely new concepts, but they have previously been inadequately brought together for the handling of permanent data.

The concept of virtual storage has been joined with a system-managed storage hierarchy.

The application model of data is supported by data in virtual (DIV), which was introduced in MVS/XA. The introduction of the new architecture and the accompanying high-level language support through data window services combine to greatly increase the potential value of data in virtual for the application programmer. Because data in virtual provides the primary I/O support for the new virtual addressing facilities, it is important to understand the basic functions provided. The next section describes data in virtual, and following sections explain the way in which it relates to the new architecture.

Data in virtual

Data in virtual is a system service that allows applications to work with the contents of permanent DASD datasets as though the entire file actually resided in virtual storage. Data in virtual was first made available in MVS/XA SP 2.2. (See References 2 and 3.) The

concept of implicitly accessing the contents of a permanent dataset by referencing virtual storage has had previous implementations. The IBM time sharing system (TSS) supported the concept on System/360 Model 67, and there are other examples in the data processing industry. One of the new features is the use of more than one level of storage hierarchy in an addressability environment where large quantities of virtual storage can be dedicated to a user. Virtual addressability combined with large amounts of processor storage managed at multiple levels makes more practical the addressing of permanent data directly through virtual storage than had been possible in the past.

Data in virtual is based on virtual storage and the relocation functions in the processor that MVS uses to manage the resources required by the operating system and application programs. At any given time, a virtual storage page can reside in a main storage frame, an expanded storage frame, or in a DASD paging dataset slot. When a virtual page is referenced but not backed by a main storage frame, a *page fault* occurs. A page fault causes an address-translation exception. Therefore, the operating system must find and allocate a main-storage frame and retrieve the contents of virtual storage from expanded storage or paging DASD.

Data in virtual adds another dimension to the meaning of virtual storage, because it permits the application to relate a virtual-storage range directly to the contents of a permanent DASD resident dataset. The particular form of permanent dataset supported is a new VSAM format, called a linear dataset. Ordinarily, a virtual page is related to DASD storage only when the contents of the virtual page have been paged out. When ordinary virtual storage is freed, the paging DASD space is freed also. A data-in-virtual object remains on permanent DASD regardless of the state of virtual storage. System services are provided to establish the relationship of virtual storage to a permanent DASD-resident dataset, but no data are read until an application reference causes a page fault. The real storage manager (RSM) component of MVS recognizes that the virtual storage is related to a permanent DASD dataset and causes the appropriate data to be read. At some later time, when the application updates the dataset, a SAVE can be requested that writes only the changed pages to the permanent DASD. The reads and writes are managed with special block-processing routines, instead of the VSAM routines used with normal GET/PUT processing.

Note that the only pages read are those referenced, and of those, only those that have changed have the potential to be written. This is quite different from what would occur if the same application capability were attempted with the conventional I/O model. The entire dataset would have to be read, causing it ultimately to end up replicated in paging storage. Unless the application keeps a record of the part of the dataset that is changed, the entire dataset might have to be written when the application does an update.

Data-in-virtual datasets are usually referred to as *data-in-virtual objects* to distinguish them from conventional datasets with an access-method-imposed format. Data-in-virtual objects have only the format defined by their creators and users. There is no control information imbedded in the object by the operating system. The application is permitted to define whatever data structure suits its needs. The only rule is that the maximum size for a single data-in-virtual object is 4G bytes. Relative byte addressing can be used to move from one location in the object to another. Index structures of any size and shape can be defined. The application interface is totally device-independent so that data-in-virtual objects are completely portable across DASD types. It makes no difference whether the data in the object is sparse or dense. Only the blocks corresponding to the pages referenced by the application in the address space are read from DASD, and only pages that have been changed are written to DASD. The physical block size used for the objects is 4K bytes, because of the relationship with virtual storage paging. This block size is visible to the application only in that the virtual view must begin and end on 4K-byte boundaries. There are no restrictions on data structures spanning physical blocks.

The application interface is totally device-independent.

The user of data in virtual is able to work with the application model of the data without the constraints of trying to conform to the rules of an I/O model.

The application can relate large virtual storage areas to the dataset and cause a working set of the dataset to be available in processor storage, thus avoiding I/O operations for rereferenced data. Efficiency is achieved by reducing the amount of data read and written to DASD. Using data in virtual for sequential processing has been improved in ESA/370 by the addition of a read-ahead capability that reads multiple pages optionally. This option is available for assembler language programs that use data in virtual and for high-level language programs that use data window services.

A data space is a capability available to programs that use the new architecture directly.

Applications that make repeated references to areas within their data and that update scattered locations run very efficiently with data in virtual. Many applications that try to keep large amounts of permanent data in virtual storage, using conventional access methods, see a substantial improvement when data in virtual is used.[4] On the other hand, an application that reads an entire dataset and inserts new records to create a new dataset is not a good candidate, unless the dataset is restructured to contain sufficient voids to contain the insertions.

Data in virtual consists of the following services:

- IDENTIFY & ACCESS—OPEN the dataset.
- MAP—Define the virtual storage range and block offset in the data-in-virtual object that are to be related. No data are read at this time. Data are read at the time the application references the virtual storage and then only the referenced data are read. MAPs may be for as little as a single page and as large as all the available virtual storage in the address space.
- SAVE—Write the changed pages to the data-in-virtual object. The real storage manager (RSM) detects which virtual storage pages differ from the version on DASD and causes only those pages to be written.
- RESET—Discard all changed pages. This causes the view of the object to be restored to the state of the last SAVE or last ACCESS, whichever is the more recent. Unchanged pages that are in main or expanded storage remain there.
- UNMAP—Ends the relationship between virtual storage and the data-in-virtual object.
- UNACCESS & UNIDENTIFY—CLOSE the object.

When data in virtual was originally released, assembler was the only user language supported by the interface. It was envisioned that applications would use data in virtual by invoking assembler language subroutines that worked with the data-in-virtual interfaces to set up mappings and save the changes. The applications would continue to be written in the high-level language of choice, and the applications would reference virtual storage to obtain the data and make the desired changes directly. Nevertheless, it was recognized that data in virtual could be more easily used by more applications if there were some support for high-level languages. This support came through VS FORTRAN, which allows calls to FORTRAN library subroutines that interface with data in virtual. VS FORTRAN was announced in November of 1987.

New addressing constructs

This section discusses new virtual addressing capabilities in MVS/ESA™ and the way in which they are supported in data-in-virtual services. ESA provides new options and added flexibility for direct users of the architecture. It also makes new functions and capabilities available to applications that have not been changed to run in the new addressing mode. A data space is an example of capability available to programs that use the new architecture directly.

Data spaces. A *data space* is a new addressing entity introduced in MVS SP 3.1.0 as part of ESA application enablement.[5] In contrast with an address space that contains system programs and data in addition to the user programs and data, a data space contains only data. Programs may be stored in a data space as data, but they cannot be executed in a data space. Data spaces may be as large as 2G bytes and are usable in their entirety by the application. The IBM 3090E implementation restricts the use of the initial 4K bytes. This restriction is made largely transparent by the data-space-create service, which returns a data-space origin of 0 or 4K bytes depending on the CPU. There is no storage reserved for common area or any system control blocks. Even the segment and

page tables needed to manage virtual storage reside elsewhere in other storage controlled by the RSM.

Data spaces are created by means of a new RSM data-space service macro that is invokable by problem state programs. Authority to access a data space is controlled by hardware, so that, in addition to increasing virtual storage addressability, data spaces improve integrity by permitting data isolation. In a multitasking environment within an address space, an individual problem program subtask can create a data space and all other tasks can be prevented from altering or retrieving data in that data space. Any or all of the subtasks could have their own data spaces. Authorized tasks can selectively share access to data spaces they own with other tasks in the same address space and with tasks in other address spaces.

Data spaces can help eliminate the requirement for the remappings and the management of virtual buffers.

Within the system, there exist multiple address spaces, and within each address space there can exist multiple tasks. Each of the tasks can own multiple data spaces. Overall, the virtual addressability available to an application is limited only by the bounds imposed by the hardware configuration.

Applications written in assembler language and running in the new addressing mode can access data spaces directly through the full set of System/370 instructions that access and manipulate storage.

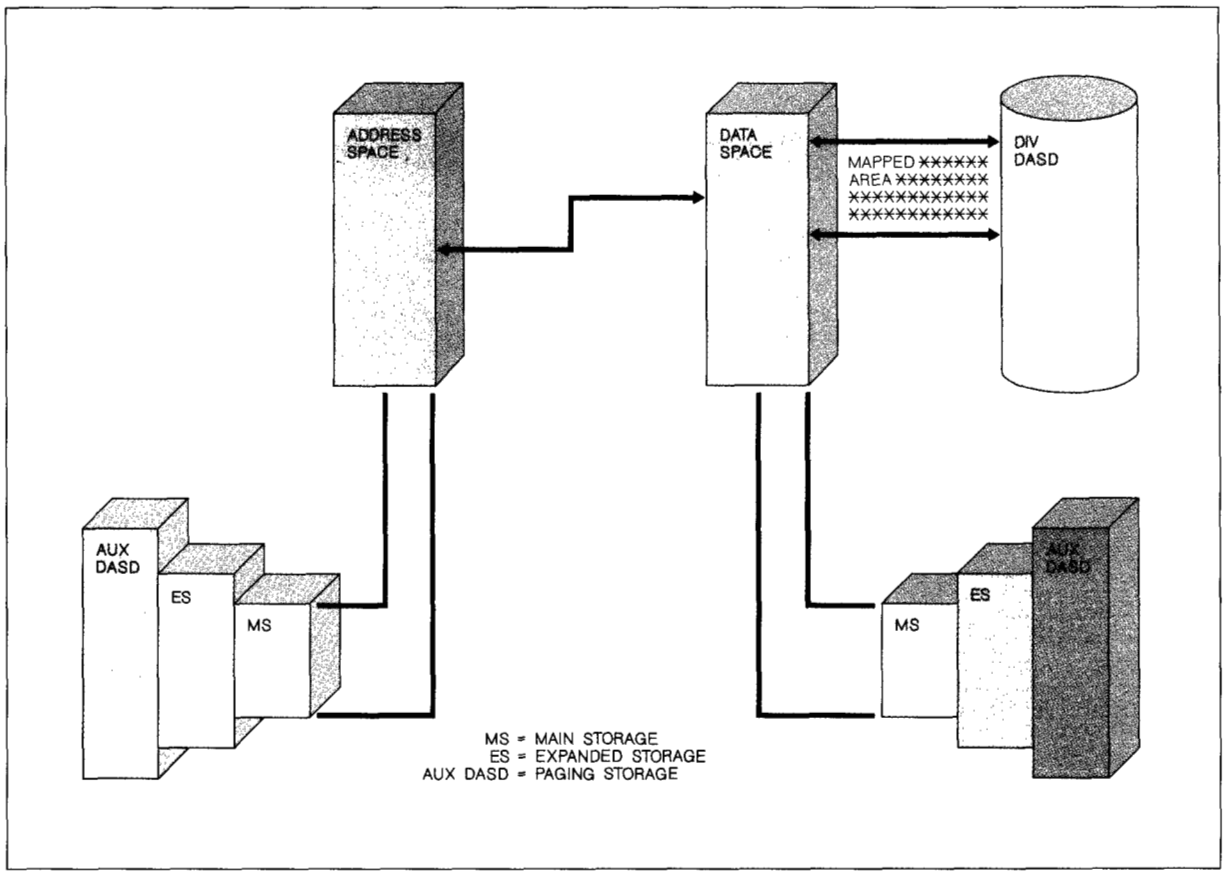
Data in virtual has been enhanced to make it possible to MAP data-in-virtual objects in data spaces. Users who run in the new addressing mode can set up relationships between data spaces and permanent data-in-virtual objects using the same techniques as they would for their address space private area. Data in virtual provides the means for applications to accomplish I/O between a data space and DASD directly, without moving the data through an address space. All the functions of data in virtual are available to the creator of the data space. Figure 1 shows the relationship of the address space, the data space, the data-in-virtual object, and the paging hierarchy.

Data-in-virtual objects can be as large as 4G bytes. In MVS/XA, the data-in-virtual user of large objects or perhaps a number of smaller objects may have been unable to keep in a MAP all of the active data. The only remedy was to remap the virtual storage to be able to access all the data. This necessitated managing the mapped storage much the same way buffers are managed for conventional I/O. Data spaces can help eliminate the requirement for the remappings and the management of virtual buffers. A single, large data-in-virtual object can be mapped across one or more data spaces or several smaller data-in-virtual objects can be mapped within a single data space. Data spaces thus provide the capability for very large amounts of permanent data to be continuously available for reference within the virtual storage owned by the application.

Virtual lookaside facility. In addition to providing application access to data spaces, MVS/ESA has added services that use data spaces to provide improved performance for users of the operating system. Virtual lookaside facility (VLF) is an MVS component that uses data spaces for storing and retrieving named objects. Data are stored in the VLF data spaces on byte boundaries and can be retrieved into the requester's address space on byte boundaries. The VLF data spaces are managed as part of the MVS paging hierarchy, and the data-space pages may reside in main storage, expanded storage, or auxiliary paging storage. VLF provides a set of easy-to-use, high-performance services that can be invoked by authorized subsystems or major applications to provide a virtual storage *lookaside*. Conceptually, a lookaside provides an alternate, higher-performance means of accessing data, by keeping it in a more readily available type of storage.

The *primary* intent of VLF is to enable components that repeatedly retrieve high-usage named data (such as partitioned dataset [PDS] members) on behalf of many users in the system to avoid I/O operations. Response time can be improved by maintaining frequently used objects in virtual storage without requiring any change in the application. Each component using VLF is responsible for obtaining the individual named objects stored in VLF from DASD by whatever means is appropriate for those objects.

Figure 1 DIV object mapped to a data space



The data to be stored in VLF are presented by the requesting component in the form that the component wishes to retrieve the data. Given this, it can be seen that VLF can be used for a variety of types of data stored in any form required, as long as a suitable naming scheme for the objects exists. The data can be derived from any source within the system, not necessarily from DASD. However, when the named VLF objects correspond to PDS members, VLF provides additional support to assist in the maintenance of members.

VLF objects and naming structure. The data to be stored in VLF are structured into groups known as *classes*, each of which represents data managed by a different component or authorized application. Within each class, every *object* has two levels of name associated with it. The *major name* specifies a subgroup of objects within a class, and the *minor name* specifies a specific object within a group. Within a class, each major name must be unique. Within a major name, each minor name must be unique. Thus, for a given class having several major names, multiple objects may exist with the same minor name.

The naming structure mimics the existing structure used to access members of partitioned datasets. The major name is functionally analogous to a concatenation of the volume serial and partitioned dataset name, i.e., it uniquely identifies a group of objects. The minor name is functionally analogous to a PDS member name in that it uniquely identifies a specific data object by name.

Applications that can benefit by using VLF are those for which there are multiple users of the data or the data have a high frequency of reuse. Data stored in VLF are subject to page stealing, just as data stored in a user's address space or data space. Therefore,

appropriate data must have a high enough rate of usage that they are likely to stay in real or expanded storage. VLF is better suited to relatively small objects, because less virtual storage is expended to save the I/O. Very large objects when not used frequently enough to remain in real or expanded storage may take longer to retrieve from VLF than by doing traditional I/O to DASD.

The component or authorized application that requires VLF need interface with only a few macros to share large storage capacity among all users of a class. Within the system, the users of the class have fast retrieval of named objects while being assured of the integrity of the data.

VLF enables authorized installation programs as well as IBM subsystems and system components to maintain named objects in virtual storage and retrieve them rapidly on behalf of many end users. Objects can be deleted and replaced by new versions, but the copy in VLF storage cannot be modified.

Library lookaside. Library lookaside (LLA) dynamically selects and stages load modules into a virtual lookaside facility (VLF) data space to avoid program fetch I/O. LLA also keeps copies of the directories of installation-designated libraries in its address space to eliminate library directory search I/O. The LLA copies of library directories can be selectively refreshed to activate new versions of library members in a controlled manner. LLA can be used to improve library performance, system usability, and system availability.

In order to use LLA, the installation must define the LLA class in the VLF parameter library (parmlib) member and list the libraries in LLA's parmlib member(s) that LLA is to manage or must not manage. LLA manages the linkage list (LNKLST) libraries by default. The installation can also dynamically add and remove any cataloged libraries for LLA. VLF and LLA are both started tasks. If LLA is not defined to VLF or VLF is not started, LLA will not cache-load modules.

While LLA is managing a library, the LLA directory eliminates search I/O for that library and conceals all updates to that library's directory on DASD. Copies of the directory entries in the LLA directory can be selectively refreshed at any convenient time to activate new versions of the corresponding library members. LLA can be used to coordinate library updates that span multiple libraries, updates consisting of one or more parts within a single library, and multiple independent updates of both types. With LLA, it is not necessary to quiesce users before updating a library. If for some reason an update cannot be completed, LLA shields the system from the partial update and the changes can be removed without disrupting users. Users may still need to be quiesced for the short interval while LLA is being refreshed.

Data spaces provide virtual storage that can be used to avoid explicit requests that require I/O.

Another way to use LLA to control versions of library members is to update the libraries on prime shift and refresh LLA on third shift when users will not be disrupted.

For load libraries, LLA dynamically adjusts the contents of LLA's VLF storage to minimize the overhead associated with program fetch. Adjustment of content works both to add and delete modules. Load modules that become active are staged to the VLF data space, and modules that become inactive are deleted from the VLF data space. Installations can optionally redirect LLA's staging decisions by providing installation exits for LLA.

Hiperspaces. Data spaces provide virtual storage that can be used to avoid explicit requests that require I/O. Large buffer areas for data that have been read from DASD and are expected to be reused in a relatively short time can be defined in data spaces. Another alternative to DASD I/O is to use a data space for temporary work files. For these examples, where the data are stored—but not actually manipulated in the data space—there is a special form of data space which can offer greater efficiency in the use of system resources for some applications. This is particularly true if 4K-byte granularity for data storage and retrieval is workable for the application. This form of data space is called a hiperspace—for HIgh PERformance data access when compared to I/O operations from DASD. A hiperspace is created with the same RSM services used to create a data space.

The intended purpose of hiperspaces is to provide applications with the opportunity to use expanded storage as a substitute for I/O operations and thus gain very significant performance improvements. It is expected that the use of hiperspaces by applications would be encouraged where there is underutilized expanded storage capacity or where extra capacity could be installed in order to use expanded storage to avoid I/O operations.

Hiperspaces differ from regular data spaces in that main storage is never used to back the virtual pages in the hiperspace. Another difference is that data can be stored and retrieved between an address space and a hiperspace only by invoking MVS system services. The data are addressed and transferred on 4K boundaries and in 4K blocks. Because the services are part of the real storage manager, page faults are avoided and in some cases the data transfer is accomplished simply by manipulating control blocks and transferring ownership of the storage entity containing the data. The hiperspace services are available to assembler language programs, but the user of the hiperspace does not have to be in the new addressing mode required for access to data spaces, because the system services run in the addressing mode necessary to accomplish the data transfer.

Hiperspaces are used for storage and retrieval of pieces of objects, in contrast to VLF which stores and retrieves entire named objects. The hiperspace user directs precisely where the data are stored in the hiperspace, and that user must remember that location in order to retrieve the data. The VLF user does not know where the object is stored and need supply only the object name to retrieve it.

Two types of hiperspaces are supported, the *standard* data-space type for unauthorized programs and *expanded-storage-only* data-space type for authorized callers. The virtual storage in the standard type is backed by expanded storage and can migrate to auxiliary paging DASD. Just as the name implies, the expanded-storage-only type never migrates to auxiliary storage. Figure 2 illustrates how the paging hierarchy applies to data spaces and the two types of hiperspaces.

One purpose for the standard hiperspaces is as an alternative to a temporary DASD work file. When used in this way, standard hiperspaces function much as VIO to expanded storage does, except they avoid all the CPU overhead resulting from the VIO device simulation. VIO has the advantage that any application written to use one of the data management access methods that use EXCP can elect to use VIO by a simple change in the job control language.

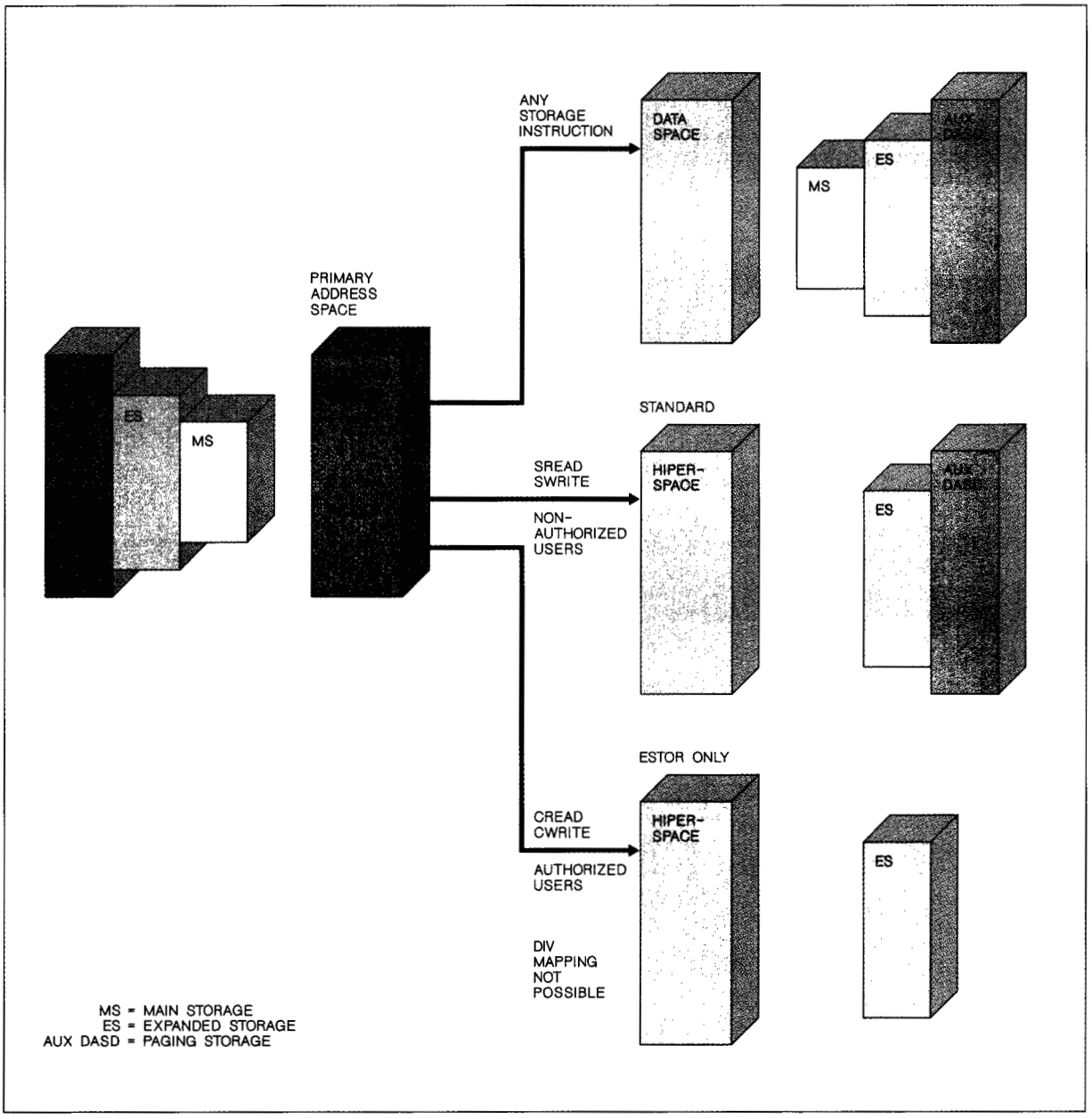
Data window services facilitate programs written in high-level languages.

Hiperspaces require change in existing applications or the development of new applications to use the new services. Data window services, which are described in the next section of this paper, facilitate this use, particularly for programs written in high-level languages. The advantage of hiperspaces is substantially less CPU overhead because the VIO device simulation and data movement required by the simulation are avoided. Hiperspaces also have greater capacity as each one can contain up to 2G bytes of data. VIO datasets are presently bounded by the size of the simulated device, which is an IBM 3380 with the capacity of 630M bytes.

Expanded-storage-only hiperspaces are expected to serve as buffers for data where there is a permanent copy on DASD. MVS can elect to steal expanded storage from the expanded-storage-only hiperspace, which means the data are no longer retrievable from the hiperspace and, therefore, must be read from DASD. Two options are available to the authorized program that must keep data in a hiperspace. One is to use the CASTOUT=NO option, which does not allow the expanded storage to be stolen. The other option is to use a *storage isolation* feature to guarantee that the working set for the owning address space does not fall below the value needed to assure backing for the hiperspace. The second option is preferred because it provides the system with more flexibility for managing system resources and facilitates such functions as dynamic reconfiguration.

The read and write services provided by RSM have attributes that facilitate efficient storage use by hiperspaces. Data written to standard hiperspaces destroy the contents of the address space pages from

Figure 2 DIV mapping and hiperspaces



which the data are written. The address space page content should be treated as unpredictable after a request to write. Given that standard hiperspaces are used for temporary storage, the design assumes that when data are written from the address space to the hiperspace the application reuses the address space pages for different data. In all likelihood, the next set of data written to the hiperspace will be built in those same pages. This allows the RSM to choose the most efficient means of doing the write operating regardless of the state of the address space virtual storage at the time of the write. In the case where an address space page is not currently backed by main storage, but instead is backed by expanded storage or auxil-

iary DASD, the expanded storage or auxiliary DASD slot is taken from the address space and given to the hiperspace. The data are in effect moved to the hiperspace by simply manipulating pointers in control blocks rather than physically relocating the data. This transfer function is also available as an option for expanded-storage-only hiperspaces when the address space page is backed by expanded storage.

When there are no expanded storage frames available at the time of a write request to a standard hiperspace, RSM directs the write request to auxiliary storage rather than cause additional migration from expanded storage to auxiliary storage. When this occurs, the real frames are taken from the address space while the I/O operation proceeds and completes. In the meantime, the caller is resumed and can begin to move the next set of data into the virtual storage, because it now looks like freshly acquired storage. The pages will be backed with new real frames.

When data are being read from a standard hiperspace, it is possible to release the storage that is backing the hiperspace pages being read.

When data are being read from a standard hiperspace, it is possible to release the storage that is backing the hiperspace pages being read. This is recommended when an application knows that the data are not going to be reread, which is frequently the case with work files. This saves the overhead of managing unnecessary migration to auxiliary paging storage.

Data-in-virtual objects can be mapped to standard hiperspaces just as they can for address spaces and data spaces. The difference is that instead of referencing the mapped page directly, the application invokes the RSM service to retrieve or update the desired page or pages in the mapped area in the hiperspace. In such a case, RSM notes that the hiperspace pages are mapped to a data-in-virtual object and reads them directly from DASD to the address space pages. When the application is ready to work with some different data, the current data can be captured in expanded storage backing the hiperspace. When the data are needed again, they need not be read from DASD. By this means, data-in-virtual users who are not using the new addressing mode can buffer data in expanded storage even when their address space is exhausted. Data in virtual in a hiperspace is also used internally by MVS in the implementation of scrolling support in data window services as discussed later in this paper.

VSAM use of hiperspaces. Users of VSAM local shared resources have the option of electing a second level of buffering, in addition to the VSAM virtual buffer pool in the address space. The VSAM buffer definition function is expanded to specify the number of hiperspace buffers to be obtained, in addition to the conventional address space buffers. The buffer size must be a multiple of 4K bytes. The number of hiperspace buffers are expected to be much greater than the number of address space virtual buffers because hiperspaces use a less expensive resource, both in terms of dollar cost and virtual storage constraint.

When the user of VSAM requests a record, the VSAM buffer management routine checks both the virtual buffer pool in the address space and the hiperspace to determine whether the request can be satisfied without requiring an I/O operation. If the record is found in the virtual buffer pool, it is given to the requester. If the record is found in the hiperspace, a virtual buffer is made available by writing to the hiperspace the contents of the virtual buffer that has been unreferenced for the longest time. The hiperspace buffer is then read into the newly available virtual buffer in the address space. The hiperspace buffer just read is now available to be written to by the next request. When the record is not found in either the virtual buffer or the hiperspace buffer, a virtual buffer is needed for the I/O operation. The VSAM buffer manager selects the virtual buffer that has gone unreferenced the longest and writes it to the hiperspace buffer. The hiperspace buffer selected is either the last buffer read, or, when those are all used, the hiperspace buffer that has been longest unreferenced is selected.

In summary, hiperspaces provide services for authorized and unauthorized assembler language pro-

grams to direct data to expanded storage as an alternative to I/O operations to DASD. Because MVS services are used to explicitly define the application request, RSM can achieve efficiency in resource management that is not possible in the normal management of virtual storage.

Data window services

As noted in the discussion on data in virtual, new system services must be made accessible to programs written in high-level languages. New system services, all callable from a broad array of high-level languages, have been defined to facilitate application exploitation of the vast expansion of addressability realized by ESA. Data window services combine the facilities of data in virtual with hiperspaces to provide new capabilities for applications working with temporary and permanent datasets. Data window services are callable from user applications written in FORTRAN, COBOL, PL/I, PASCAL, and assembler language, using standard linkage conventions. Data window services give the programmer working with a high-level language access to new ways of dealing with data that can simplify the application and improve performance, largely by reducing the number of I/O operations.

Data window services extend the capability to do basic data-in-virtual functions in the address space to all the supported high-level languages. In addition, functions are added that are available to data-in-virtual users only on MVS/ESA. Dynamic allocation and creation of permanent and temporary data-in-virtual objects is supported along with the capability to do scrolling and deferred writes. *Scrolling* is the capability to use virtual storage in an address space to view a portion of a data-in-virtual object and to capture the referenced and changed pages in expanded storage when moving that view to a different part of the object. When the application works with a previously viewed part of the dataset, the data are available in processor storage and can be retrieved without requiring DASD I/O operations. Data window services accomplishes this by writing the scrolled-out data to a hiperspace created by the service for this purpose. The mapping of the entire data-in-virtual object to a hiperspace and the relating of a portion of the hiperspace to the address space is shown in Figure 3.

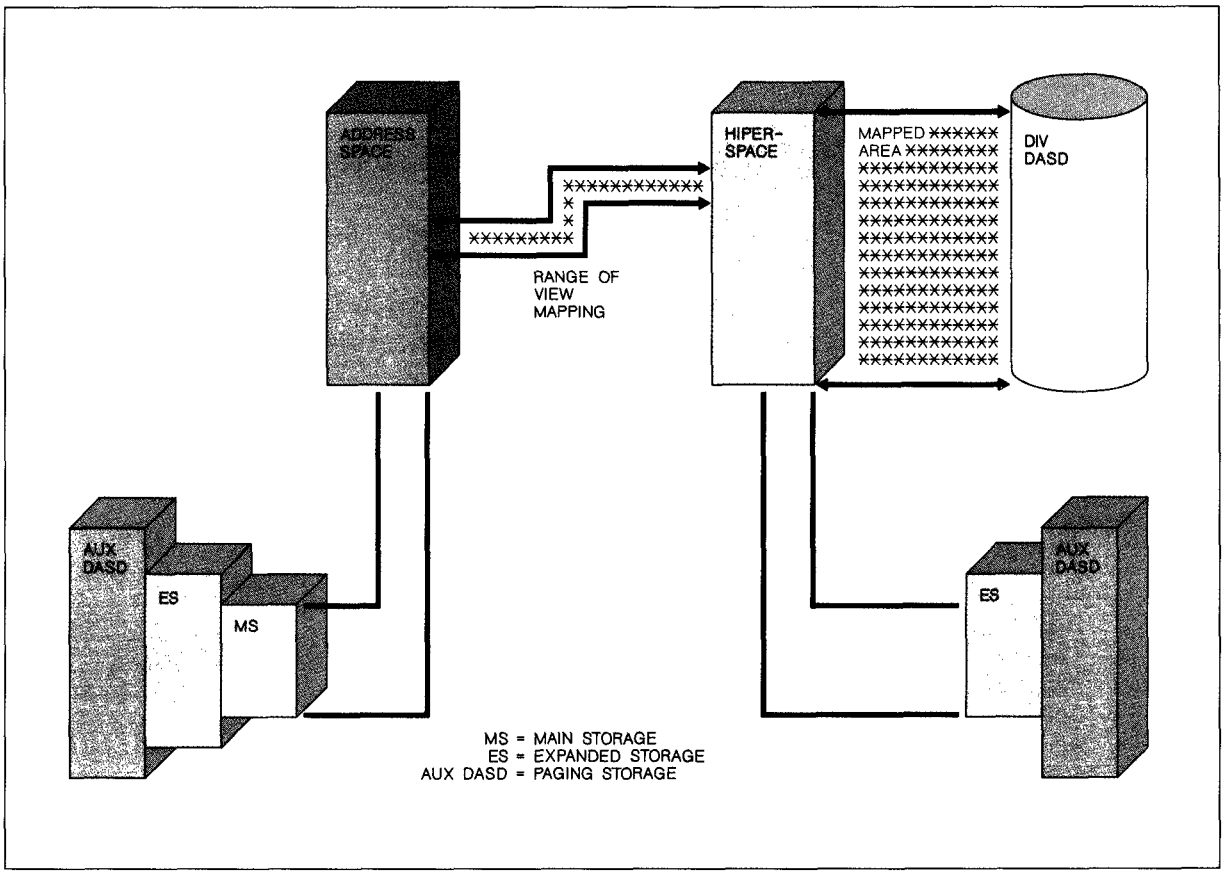
Deferred write is a data window service function that permits an application to work with a data-in-virtual object over an extended period of time, without causing the dataset to be in a transient state of partial physical modification until the process has completed successfully. This is accomplished by using scrolling to accumulate and review the changes and then using a new save capability. The new save capability writes at one time all the changed pages accumulated previously by scrolling to the permanent data-in-virtual object. Just as with direct use of the data-in-virtual interface, only the pages that are changed from the original copy on DASD are written. Pages containing data that are subject to multiple updates are written to DASD only once when using this technique. The scrolling technique is also useful for testing new applications against on-line data without requiring replication of the data in a test file.

Scrolling is also useful for testing new applications without requiring replication of the data in a test file.

Temporary objects are created by requesting the TEMPSPACE (temporary space) option in data window services and specifying the size in 4K blocks. Internally, data window services create hiperspaces as needed to contain the temporary object. When more than one hiperspace is needed, data window services manages the concatenation of the hiperspaces, although the user is unaware when the current view of the object spans hiperspace boundaries. The application is unaware of the hiperspaces and simply sets up views in the address space and scrolls out and resets the view whenever the view is to be moved to another portion of the object. The theoretical limit of a temporary object is 16T bytes, which is beyond the paging capacity of the operating system at this time. Figure 4 illustrates how multiple hiperspaces are used to contain very large temporary objects in virtual storage.

The data window services that can be called are summarized as follows, where CSR stands for callable service requests:

Figure 3 Window services view for scrolling of DIV object

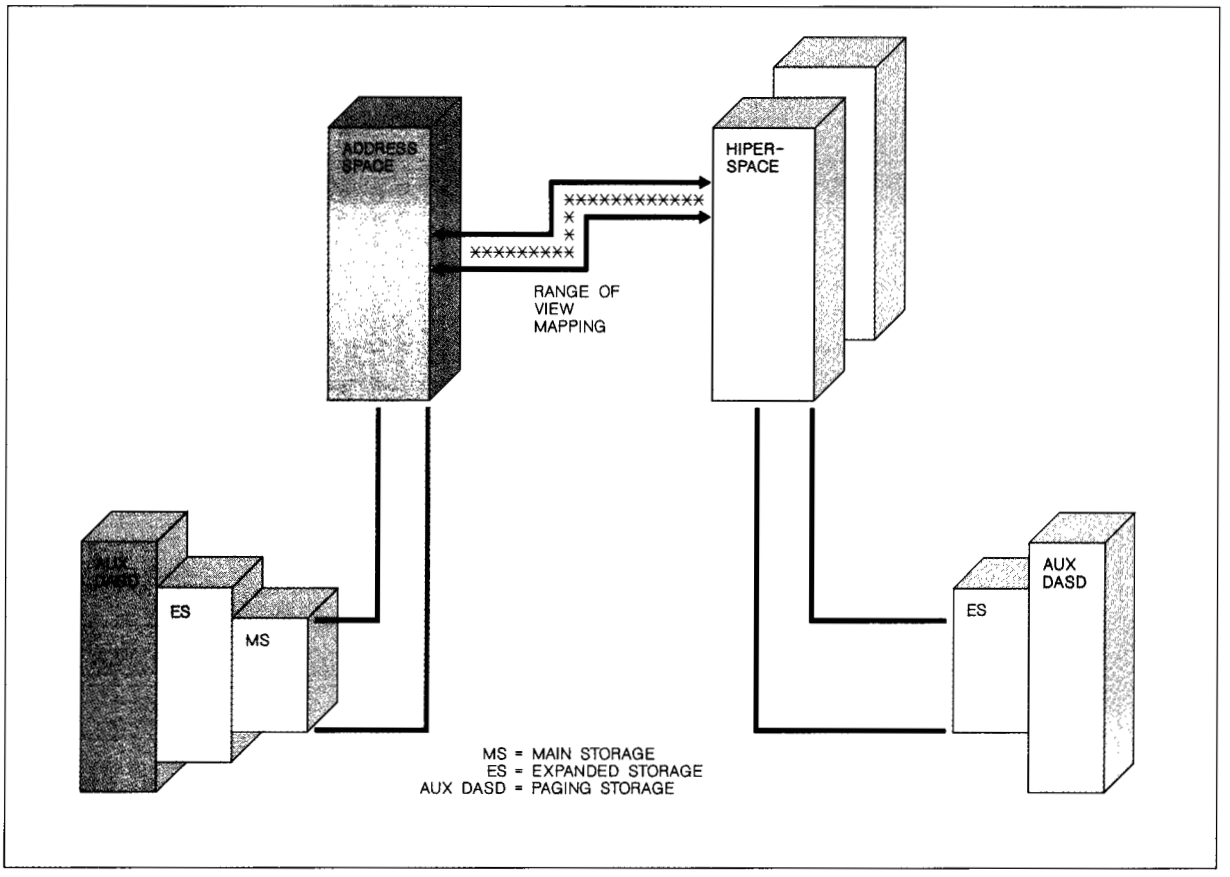


- CSRIDAC—Identify and access a linear object. Optionally create a new permanent or temporary linear object. This service is also used at the end of processing to remove access and release an identification.
- CSRVIEW—Establish a view of the linear object in virtual storage or review a scrolled-out version of a view. This service is also used to end the relationship between virtual storage and a range of data in a linear object.
- CSRSCOT—This scroll-out function captures the current view in the window with all changed pages and any unchanged pages currently occupying main storage and expanded storage for subsequent SCROLLINs or SAVEs. The scroll-out function does not update the permanent linear object.
- CSRSAVE—All changed pages in the current view and also any that have been previously scrolled out are written to the permanent linear object.
- CSRREFR—Refresh all changed pages in the current view and also any that have been previously scrolled out back to the values contained in the linear object, and discard any changes.

Concluding remarks

The Enterprise Systems Architecture has vastly expanded the virtual addressability available to system and application programs. New and enhanced system services have been made available to take advantage of the new architecture. Some of the functions will provide benefits to data processing users with little or no effort on their part. Some examples are library lookaside to reduce I/O required for program loading and VSAM hiperspaces for improved hit ratios from the buffer pool. Finally, new facilities are available for high-level language applications to provide access to new function and achieve improved

Figure 4 Window services view for temporary DIV object



performance in some circumstances through I/O reduction and efficient use of processor resources. ESA makes it possible to look at problems with perspectives that were never before achievable.

Acknowledgments

I wish to express my appreciation for the contributions made by John R. Ehrman, Donald H. Gibson, Casper A. Scalzi, and Richard J. Schmalz for their active roles in the development of the ideas that led to hiperspaces and data window services. Eugene S. Schulze was of key importance in the evolution of the hiperspace design, as was Catherine Eilert for data window services. I wish to thank Rick Reinheimer and Peter Cochrane for their assistance in preparing the material on VLF and LLA, components for which they respectively played primary design roles. I also thank Robert M. Zeliff for his constructive comments on the content of this paper.

Cited references

1. *Toward More Usable Systems*, Report of the Large Systems Requirements for Application Development Task Force, SHARE Inc., One Illinois Center, 111 E. Wacker Drive, Chicago, IL 60601 (December 1979).
2. P. R. Dorn, *An Introduction to Data In Virtual*, GG66-0259-00, IBM Corporation; available through IBM branch offices.
3. *MVS/XA, Supervisor Services and Macro Instructions*, GC28-1154, IBM Corporation; available through IBM branch offices.
4. I. L. Traiger, *Virtual Memory Management for Database Systems*, Research Report RJ3489, IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA 95120 (May 1982).
5. J. S. DeArmon, "Enhanced I/O performance via system paging facilities," *Conference Proceedings*, CMG XV International Conference on the Management and Performance Evaluation of Computer Systems, San Francisco, CA (December 1984), pp. 93–102.

6. *MVS/ESA SPL: Application Development-Extended Addressability*, GC28-1854-1, IBM Corporation; available through IBM branch offices.

Kenneth G. Rubsam *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, New York 12601.* Mr. Rubsam joined IBM in 1959. During his career at IBM, he has held various technical and management positions. One of his assignments was as team leader on development of TEST/360, an early telecommunications network simulator that was used to verify network control programs prior to their installation. He worked on the prototype for SVS, the first System/370 virtual storage operating system. He was also team leader for the data-in-virtual prototype, which was done in conjunction with a joint study with General Motors Research Division. His recent assignments include the design of the hiperspaces and data window services described in this paper. He is currently a senior programmer working in the MVS Systems Structure Design group in the Meyers Corners Laboratory. Mr. Rubsam graduated from Iowa State University in 1959 with a B.S. degree in electrical engineering.

Reprint Order No. G321-5352.