VM/XA storage management

by G. O. Blandy
S. R. Newson

The VM/XA System Product manages the vast amounts of real and expanded storage available on the new Enterprise Systems Architecture/370™ processors for both guest use and support of internal operating system functions. The management algorithms are examined, and the rationale for their selection is presented.

The new Enterprise Systems Architecture/370™ (ESA/370™) processors provide a user with unprecedented amounts of real and expanded storage. The exploitation of this storage by the Virtual Machine/Extended Architecture System Product™ (VM/XA SP™) is unique as, indeed, is its role as a control program.

VM/XA SP manages the resources of a System/370 Extended Architecture (370-XA) system or ESA/370 system to create multiple virtual machines, each capable of running an operating system such as Virtual Storage Extended (VSE), Conversational Monitor System (CMS), or Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA™). Each virtual machine may be viewed as an instance of a complete processor complex with features, facilities, and resources as rich or even richer than the native complex being managed by VM/XA SP.

VM/XA SP is designed to exploit many of the extended addressing capabilities introduced with 370-XA. This exploitation allows operating systems such as MVS/ESA to utilize the full architectural capabilities when running in a virtual machine environment. VM/XA SP also uses many of the 370-XA addressing capabilities to support control program functions and to provide special facilities for CMS virtual machines such as the minidisk caching feature discussed in a related paper[1] in this issue. Here we focus on the exploitation of very large real, virtual, and expanded storage by VM/XA SP and describe in detail the mechanisms and algorithms employed to support these capabilities efficiently.

The two main sections that follow explore two broad categories of VM/XA storage management. The first section describes how VM/XA SP uses the native storage hierarchy of real, expanded, and auxiliary storage to create large virtual machine storages and provide for the use of dedicated expanded storage by a virtual machine. The second section concentrates on the use of virtual and real storage by VM/XA SP for its own internal storage requirements.

Virtual machine storage support

Each virtual machine is the functional equivalent of a real processor complex. VM/XA uses the techniques of time-sharing, partitioning, and dedication to manage native resources in such a way that virtual machines appear to have their own CPU(s), vectors, real storage, expanded storage, I/O devices, etc. A combination of hardware facilities, software constructs, and system resources enable VM/XA to manage virtual images of real and expanded storage.

Virtual machine real storage. Running on a large ESA/370 complex such as an IBM 3090S, VM/XA can support as many as several thousand virtual machines whose combined "real" storage sizes can be orders of magnitude greater than the real storage capacity of the native processor. At the same time, special preferred virtual machines with fixed real storage can approach native performance.

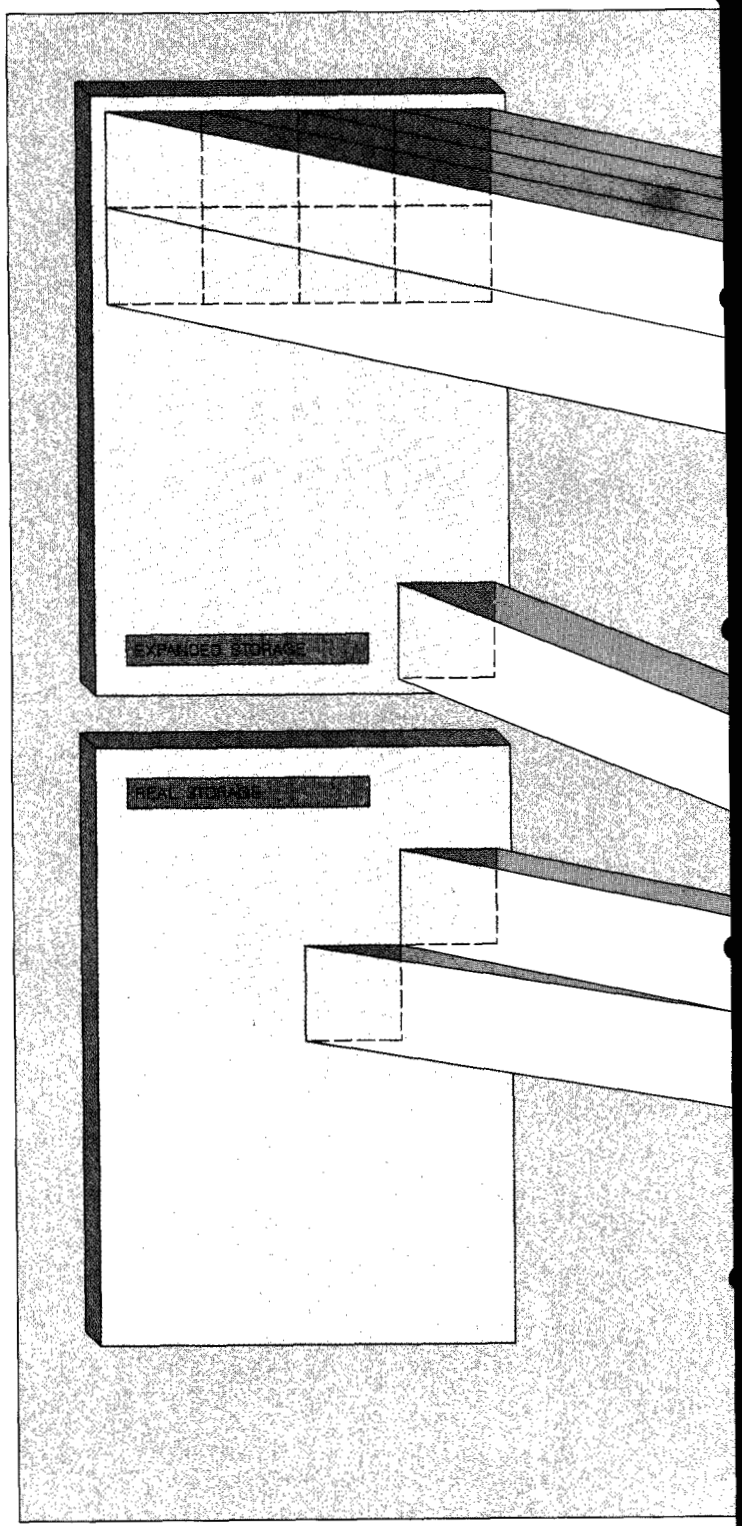
System/370 Extended Architecture introduced a significant hardware feature used to maintain virtual machines. The interpretive-execution facility, which has been described by Gum,[2] provides a means for establishing a virtual machine environment and controlling the execution of programs running in that environment. To invoke the facility, VM/XA executes the Start Interpretive Execution (SIE) instruction, specifying, as an operand, a control block known as the state description. The state description defines the architecture, facilities, and state of the machine to be interpreted. The execution of the SIE instruction causes the CPU to enter interpretive execution mode and to execute instructions under control of the state description.

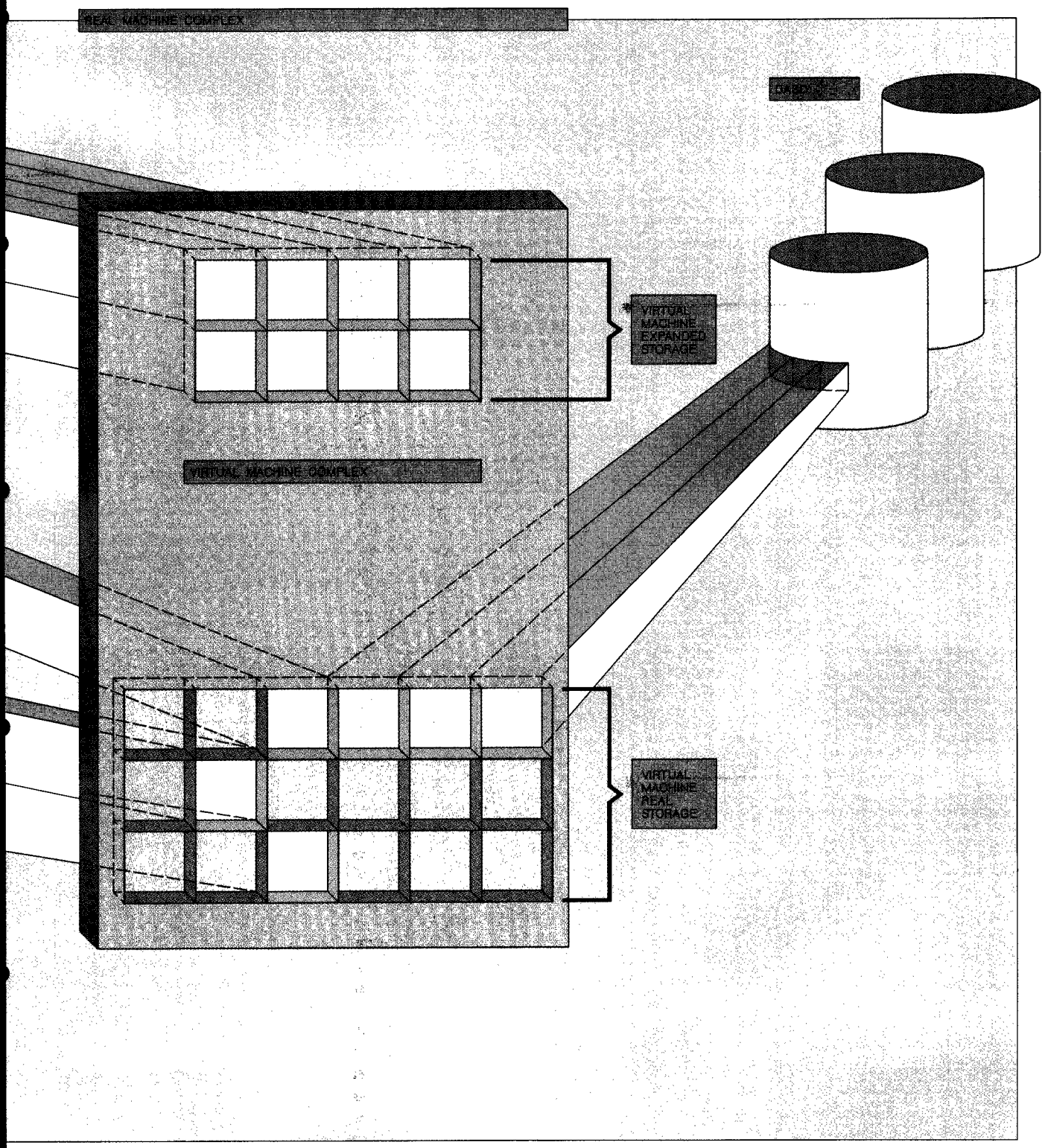
The interpreted virtual machine is known as a guest, whereas the real processor on which VM/XA is running is known as the host. The terms *guest* and *host* are also applied to the programs running in the respective machines. For the purposes of this paper the term *host real storage* identifies physical storage belonging to the native complex. *Host virtual storage* identifies storage defined by VM/XA-maintained page and segment tables. The term *guest real storage* refers to the storage which the guest perceives to be the physical storage belonging to its processor complex.

Interpretive-execution mode, also known as emulation mode, is capable of handling many but not all of the architectural requirements of a virtual machine. When a situation is encountered that cannot be handled in emulation mode, the CPU will return control to the host program (VM/XA SP), providing required status information in either the state description or prefix page fields.

There are two methods of defining guest real storage when operating in interpretive-execution mode. In the preferred storage mode, a contiguous partition of host main storage is reserved to represent guest real storage. This storage is fixed so that no paging is performed by the host. However, the preferred storage mode may be used for only a small number[3] of guests, and the size of preferred virtual machines is

Figure 1 Pageable-mode guest with dedicated expanded storage




REAL MACHINE COMPLEX
VIRTUAL MACHINE EXPANDED STORAGE
VIRTUAL MACHINE COMPLEX
VIRTUAL MACHINE REAL STORAGE

limited to the amount of real storage available minus the amount required by VM/XA and other preferred guests.

In pageable mode, guest real storage is represented by the host virtual storage. VM/XA maintains a separate address space for each pageable virtual machine using the segment table and page table structures of the dynamic address translation (DAT) facility.

VM/XA maintains a separate address space for each pageable virtual machine.

The advantage of pageable mode is that all of guest real storage need not be resident in host real storage. To operate efficiently, a guest need only have access to that subset of storage that the guest program is currently referencing. The remainder of guest real storage may reside on expanded storage or a direct-access storage device (DASD). The page table entries for nonresident pages will be marked invalid, and any attempt by the guest to reference these entries will result in a host page fault. Emulation mode will be exited and control will be returned to the host via a program interrupt. Once the page is brought into host real storage, the execution of the virtual machine can be resumed. A pageable-mode guest with dedicated expanded storage is shown in Figure 1.

In addition to the control structures defined by architecture (segment tables, page tables, state descriptions), an additional set of control program fields are required for the maintenance of guest storage. Two control blocks, each residing in a 4K frame of host storage, are of special interest:

1. The Virtual Machine Description Block (VMDBK) is the primary control block used to define and control the virtual machine. The VMDBK contains both information required by the hardware (the state description is imbedded in the VMDBK) and information used solely by the control program for virtual machine maintenance. A section of this control block is dedicated to the management of guest real storage. This section contains statistical information and pointers to the various lists and structures required to represent guest real storage. Also within this section are a number of software "locks" that are used to serialize access to these lists and structures.
2. The Paging Management Block (PGMBK) defines a single segment of guest real storage. Like the VMDBK, it contains both information used by hardware (the page table is imbedded in the PGMBK) and information used solely by the control program. The PGMBK contains information pertaining to the location of nonresident pages (i.e., expanded storage and auxiliary storage locations), the guest's view of the storage keys, and various status indicators.

The segment table created by the control program (CP) for the guest real storage must be page-aligned for architectural reasons. However, many virtual machines, such as those that run applications under CMS[4] (Conversational Monitor System) require only a few megabytes of guest real storage. For these machines, it would be highly wasteful to allocate a full page for each segment table when only a few words are required. To prevent such waste, the VM/XA design allows the segment table to take two forms. In one case the segment table is packaged as part of the VMDBK. This combination will support virtual machines with storage up to 32 megabytes. For virtual machines that require larger storage, a separate, page-aligned segment table is provided.

Very large virtual machine storages may be defined, but in a given session, the guest program may reference only a fraction of this storage. To achieve efficiencies in real and auxiliary storage usage, VM/XA dynamically allocates PGMBKs only when a segment is referenced. Within each segment, auxiliary pages and real frames are required only as each page is referenced.

The two storage modes provided by the interpretive-execution facility allow installations to run a few high-performance preferred guests and numerous pageable guests. Although the preferred mode of storage, which uses host real storage to represent guest real storage, necessarily limits guest real storage to a size smaller than the native complex, the pageable mode, which uses virtual storage to represent guest real storage, has no such limitation. Pageable virtual machine storage differs from native real machine storage in only two significant ways:

1. Because the entire storage hierarchy (real, expanded, and DASD) may be used, access to a storage location may be delayed until the referenced page is made resident in real processor storage.
2. It may be larger than the amount of storage available to the real processor.

Management of host real storage to support guest real storage. A subset of guest real storage must be resident in host real storage to allow a virtual machine to execute. This subset must include those instructions of the guest program that are currently executing and any data referenced as operands of those instructions. Because the real storage of preferred-mode guests is always host-resident, VM/XA is relieved of the task of managing this storage. However, for pageable-mode guests, it is the responsibility of VM/XA to assign host real storage frames as required.

Host real storage frames are either permanently assigned or are available for dynamic allocation. Frames permanently assigned include those containing the resident portion of the CP nucleus and those dedicated to any preferred virtual machines. The remainder is available for dynamic allocation and is managed by the VM/XA real storage manager (RSM) to maintain virtual address spaces and free storage.

Each 4K frame of host real storage is represented by a four-word (16-byte) frame table entry (FRMTE). The first two words are used to chain the FRMTEs on various lists using forward and backward linkages. The remaining fields are used to describe the status of the frame and to point to the page table entry for those frames that make up the resident portion of a virtual storage address space. Figure 2 illustrates the FRMTE in the storage control block structure.

The "available list" is the heart of the real storage manager and consists of a queue of FRMTEs representing frames that are immediately available for allocation. For smooth system operation, RSM must ensure that there is always an adequate supply of available frames. Initially, all nonpermanent frames are placed on the available list.

When a pageable virtual machine first logs on to the VM/XA system, the guest real storage is logically "empty" (all binary zeros), and no real storage frames are required to represent guest storage. As the virtual machine initiates its operation, it will reference pages within its real storage. Each initial reference will cause a page fault that will be processed by the VM/XA control program. A real frame will be obtained from the available list, its address will be stored in the page table entry (as required by hardware), and the FRMTE representing that frame will be added to the queue of FRMTEs anchored in the VMDBK representing that virtual machine. For a guest real page which has never been referenced before (a first-time page fault), CP will merely allocate a frame and clear it to binary zeros. Guest execution will be resumed. This operation will continue for one or more guest virtual machines until the supply of frames on the available list falls below a predetermined level. At that point frames must be "stolen" from another (or perhaps the same) virtual address space to replenish the available list supply. The page table entries for the stolen pages are marked invalid, and the contents of the frames are paged out to expanded or auxiliary storage.

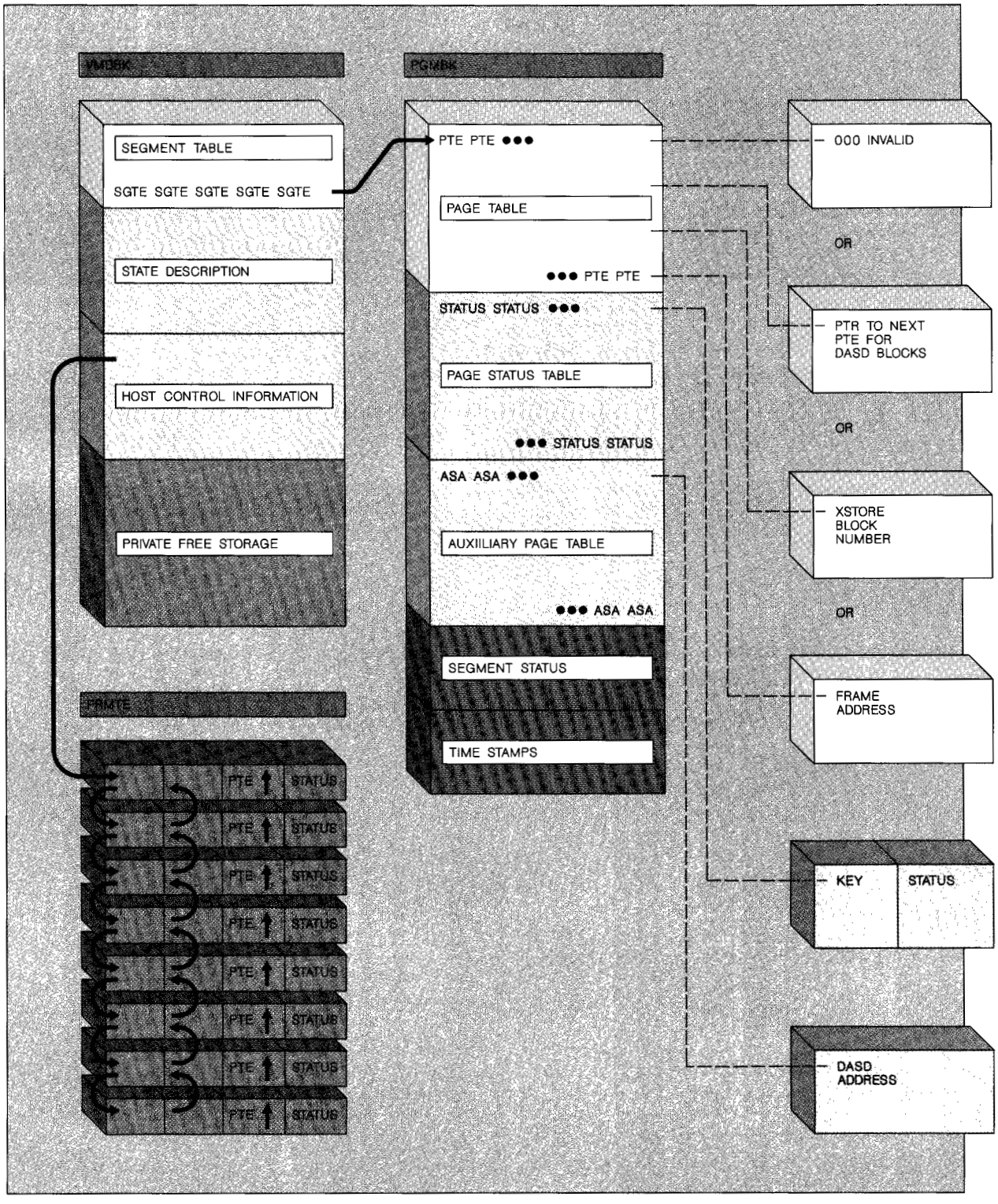
When a guest references a page that had previously been paged out, a page fault will occur. A frame will still be needed from the available list, but instead of clearing it to binary zeros, the page will be brought in from its assigned auxiliary storage location. The external storage location may be an expanded storage block, a DASD page, or in some special cases, another real storage page. Once the page fault is satisfied, the guest is eligible to resume normal operation.

The selection of the right pages to "steal" from guests is crucial. Unfortunately, this selection can only be a best guess in any operating system. VM/XA employs a method of trying to determine the least recently used pages and selecting them to be made available. In order to determine which pages a guest machine is referencing, the hardware reference and change indicators are used. VM/XA periodically executes a reorder function which uses the RRBE instruction to detect referenced frames and reset reference bits so that subsequent references may be placed.

The purpose of the reorder function is two-fold. First, it identifies unreferenced pages to the RSM "steal" task. Second, it provides the number of referenced pages so that the working set size (WSS) of the virtual machine may be determined. WSS provides an indication of the number of resident pages the virtual machine will require to run efficiently in the future. This information is vital to the VM/XA scheduler in managing storage-constrained environments.

The reorder function is performed on a guest-by-guest basis. When the dispatcher determines that a

Figure 2 VM/XA SP storage control block structure



guest has executed for a sufficient amount of time, a reorder function is performed for that guest. This time is not fixed but is controlled by a feedback mechanism that allows a reasonable amount of time for the guest to reference enough pages to constitute a working set. Many virtual machines have a set of pages that are always referenced, such as in the guest's resident nucleus, and a set of pages that vary in their reference pattern. This varying reference pattern is entirely dependent upon the nature of the guest transactions and cannot be predicted.

The reorder function changes the order of the queue of FRMTEs that represents the host-resident frames currently assigned to that guest. This list may contain one or more sections from previous reorder functions and a set of newly referenced frames. The output from the reorder function is a list of FRMTEs consisting of a section representing the frames that have been referenced over the interval and another section representing unreferenced frames.

The RSM function in VM/XA responds to various requests from within the system for real storage frames. These requests are satisfied by removing a FRMTE from the top of the available list and returning the address to the caller. If the number of FRMTEs remaining on the available queue falls below a predetermined low threshold, a task is initiated to replenish the available list by "stealing" frames assigned to virtual address spaces.

The steal task examines the guests that own frames and takes appropriate pages to satisfy the storage demand. The storage demand is considered satisfied and the steal task terminates when the number of frames on the available list reaches a high threshold. (Both the high and low thresholds fluctuate to accommodate varying levels of system storage demand.) While the steal task is running, requests for frames continue to be accepted by RSM. Frames are given out from the available list until there are no frames available. At this point the requesting function must be deferred. The steal task will satisfy the deferred requests as pages become available.

The selection of a guest from whom pages will be stolen is described in detail in the VM/XA publications. Briefly, the steal task will select guests for whom the reorder function has been recently performed. "Dormant" guests (those that have not recently competed for CPU resources) tend to be visited more frequently than those that are active.

Once selected, if a guest is determined to be "long-term dormant" (dormant for some system-determined period of time, typically over 30 seconds), *all* of its frames will be stolen. For all other guests, only the unreferenced pages will be taken. For active guests, this process tends to "weed out" pages that are no longer part of that guest's active working set. For the short-term dormant guest, it preserves an "interactive buffer" of recently referenced frames. The assumption is that at least some, if not all, of these frames will be required again when the guest next becomes active.

If, after visiting all recently reordered guests, enough unreferenced pages cannot be found (an unusual case), a second pass of the steal task is initiated. In this case no pages are protected, and any frame will be used to satisfy the storage demand.

Pages are also added to the available list when CMS returns real storage pages upon completion of certain functions. The CMS component is aware that it is running in a pageable virtual machine environment, and when it no longer requires the contents of certain frames, it indicates[5] to the control program that a range of pages should be released. CP removes the frames in this storage range from guest ownership and places them on the available list. Additionally, CP will release any expanded and auxiliary storage associated with the specified range.

There are some subtle interactions between RSM and the system scheduler to avoid severe over-commitment of real storage. The speed of the paging subsystem has a lot to do with the ability of the system to respond to heavy storage demands. The relationship between the reorder function, the steal task, and the system scheduler is crucial to achieve a good system balance of execution, storage demand, and responsiveness.

Use of expanded storage by VM/XA. The expanded storage facility described by Tucker[6] provides for the fast synchronous movement of 4K blocks of data between real and expanded storage. VM/XA supports expanded storage for direct use by guests and for its own storage and data management needs.

VM/XA allows expanded storage to be divided into multiple partitions. Each partition may be dedicated to a virtual machine to be used by its guest operating system. To the guest, the partition appears to be an entire expanded storage facility having all of the capabilities that are included in the architecture. The

interpretive-execution facility provides a special mode of operation for preferred guests that permits direct access to expanded storage. For guests which do not use this mode, attempts to access dedicated expanded storage will result in an exit from emulation mode so that the host can simulate the requested function.

For its own purposes, VM/XA exploits expanded storage for traditional paging and, in a very unique fashion, to place in a cache 4K data blocks read and written by CMS. Bozman[1] discusses the minidisk-cache capability in detail. We now discuss the design for support of guest real storage.

The VM/XA design treats expanded storage as an extension of real storage. As such, a page is in either real storage or expanded storage, never both. Because of this treatment, VM/XA is able to use the page table entry (PTE) to hold the expanded storage block number when the contents of that guest real page reside in expanded storage. The 370-XA architecture provides for a full-word PTE which is large enough to hold the largest possible expanded storage block number.

Obviously, if a PTE contains an expanded storage block number, the PTE cannot also be valid for translation. For this reason, the invalid bit is always placed on in the PTE when it is occupied by a block number. The value of the corresponding bit for the block number is placed in an auxiliary control word in the PGMBK associated with the PTE along with other flags indicating the status of the page. The VM/XA design allows for four possible states in the PTE:

1. Real-frame-address all zero with invalid bit on. The page has never been referenced, or the page is paged-out to a single DASD page location.
2. Real-frame-address nonzero with invalid bit off. The page is valid and may be used to translate a guest real address to a host real address.
3. Expanded-storage-block-number with invalid bit on. The page has been paged-out to expanded storage.
4. Block-paging-index with invalid bit on. The page has been paged-out to DASD along with a set of other related pages in the same segment. The block paging function of VM/XA moves a group of related pages to DASD during migration and will retrieve the entire set of pages when any one page is referenced, thus reducing overhead in the paging manager.

Before the actual page-in and page-out function used to move data between real and expanded storage is discussed, it is important to understand the allocation mechanism. VM/XA uses several control blocks to manage the allocation of expanded storage. For each megabyte of expanded storage, there is a table entry that indicates if the megabyte section is available to the hardware configuration, and if so, whether it is attached to a virtual machine or is available for CP use. Additionally, the entry identifies a bit map for allocation of individual blocks within the section. This bit map exists even for megabyte sections that are not used by CP, since if expanded storage is detached from a virtual machine, CP will use it for its own functions.

A very dense bit map is used for expanded storage block allocation.

A very dense bit map is used for expanded storage block allocation to allow for the possibility of supporting extremely large expanded storage configurations. The architectural limit for an expanded storage configuration is 16 384 gigabytes. Allowing for one bit per block, slightly more than 512 megabytes of real storage would be required to represent the allocation control block structure in VM/XA. More realistically, a 1-gigabyte expanded storage would require 16K (four pages) of bit maps. Allocation and deallocation of an expanded storage block involves setting and resetting the appropriate bit in the allocation array and translating bit displacements into expanded storage block numbers.

To speed up allocation and deallocation of expanded storage, VM/XA caches expanded storage block numbers in a 500-entry push-down stack. Initially the stack is empty and allocation of blocks is done from the bit-map array described earlier. When a block is deallocated, the block number is placed into the next entry in the stack if the stack is not full. When a block is to be allocated, the stack is examined. If it is not empty, the block number from the current entry is removed, and the stack pointer is adjusted

to the next entry. Blocks in the stack are considered deallocated even though their corresponding bit in the allocation array remains marked as allocated.

Real storage pages are paged-out to expanded storage by the real storage manager steal-task function described earlier. When the steal task selects a guest page, it requests an expanded storage block number. If one is returned, the real storage page is paged-out to the expanded storage block, and the real frame is immediately placed on the available list. Each page that is paged-out to expanded storage is time-stamped with the current time in seconds. This time stamp is in an auxiliary field in the PGMBK associated with the PTE. The migration function, described later, uses the time stamp to determine which blocks are good candidates to be moved to DASD.

Because the PTE now contains an expanded storage block number, the real frame address of the page is no longer available. Should the guest happen to immediately reference the page again, a page fault would occur, requiring resolution to another real storage frame. There is no notion of a real storage frame being reclaimed if a guest references it while it still happens to be physically resident. The cost of such a reclamation capability, although used in the past with DASD devices, is not justified with expanded storage. Even with DASD paging, little need seems to exist for reclamation if pages are read as a group.

Once a real storage page is placed in expanded storage by the steal task, it can be paged-in by the real storage allocation function. If a guest references a nonresident page, emulation mode terminates with a host page fault. A test is made to determine if the fault is due to an invalid PTE occupied by an expanded storage block number. If so, a real storage frame is allocated from a processor local queue of available frames, and the expanded storage block is paged-in and then deallocated. The PTE is made valid with the real storage address, and virtual machine execution is resumed by reexecuting the SIE instruction. The synchronous nature of the PGIN instruction allows this path to be extremely short, isolated, and exempt from normal CP dispatching. The same optimized path is used to satisfy page faults resulting from first-time page references. Here no PGIN is required, and the frame is merely cleared to binary zeros.

The processor local queue of available frames is used to avoid lock contention on the global available queue. The processor local available list is primed with entries before running the guest virtual machine. Priming permits several page faults to occur that can be satisfied from expanded storage as expeditiously as possible. Tests performed using CMS intensive environments on a 3090E processor show that VM/XA is able to sustain an aggregate expanded storage block transfer rate of over 1000 blocks per second per processor. One measurement with 5000 guests on a 3090 Model 600E with 256 megabytes of real storage and 512 megabytes of expanded storage showed a continuous paging rate to expanded storage of 4200 blocks per second. Peak paging rates of over 20 000 blocks per second have been observed in a live production environment running on a 3090 Model 300E.

Even though very large expanded storage configurations are possible, there are still many environments where the expanded storage becomes fully allocated. Full allocation can be caused by the occupancy of guest real storage pages and/or blocks containing buffers from the minidisk cache function. To maintain effective paging capacity within the expanded storage configuration, blocks are migrated to DASD.

The migration function is initiated when a request is made for an expanded storage block and the allocation routine determines that the number of available blocks has fallen below a system-defined low threshold. A migration task is initiated somewhat like the real storage steal task. The real storage steal task has a set of frames (FRMTEs) that can be examined to determine which ones to make available. No such list exists for the expanded storage block allocation. To have such a list would increase the real storage required for allocation control blocks, making it far more difficult to support very large expanded storages. Instead, the expanded storage migration function examines the PGMBKs of guests that have pages allocated to expanded storage and selects candidates for migration to DASD.

The migration function examines all guests in a round-robin fashion. Each invocation of the migrator starts from the last guest examined. For guests with pages in expanded storage, each PGMBK is examined. The PGMBK contains a summary count of the number of blocks allocated within the segment and the age (from the time stamp) of the oldest block. If the PGMBK has no blocks, the next page table is examined. This routine continues until all page tables for the guest have been examined.

The migrator uses a target age to determine whether to select a block. The target age is usually the average age of all expanded storage blocks used for host paging.[7] If a block is found that is older than the target age, it is selected for deallocation from expanded storage. Because of locality of reference, more than one block will usually be selected within each segment, thus allowing the migrator to package multiple guest pages together in a set to be written to DASD. Later, if any page in this paging block set is referenced by a guest, the entire set will be brought into real storage from DASD. Studies, including those by Kienzle et al.,[8] indicate that the reference patterns of CMS guests tend to be clustered (by guest real address) and that the page rereference rates across transactions are high. The migration selection process exploits this situation by forming sets of pages on the basis of their guest real storage addresses.

Since it is not possible to perform I/O operations directly from expanded storage to DASD, the migrator maintains a number of real storage buffers for each DASD paging device available. This number varies with system load and expanded storage use. Pages to be migrated are paged-in to the real storage buffer and immediately deallocated. When all buffers are full or an entire guest has been processed, the paging I/O manager is invoked. While that I/O operation is progressing, the migrator continues to fill any remaining buffers.

As paging I/O operations complete, control returns to the migrator. The I/O buffers are now reused for the continuing migration operation. While the migration function is running, expanded storage blocks continue to be allocated. The migration will complete only when the number of available blocks reaches the high threshold that was determined when migration was invoked.

Once migration completes, the low threshold is adjusted on the basis of allocation activity. If the number of blocks available has fallen perilously low, the threshold is raised. If several successful migrations have occurred and the number of available blocks has been sustained at an acceptable level, the threshold is lowered. Note that this feedback mechanism results in a quick response to surges in expanded storage demand but a cautious response to dwindling demand.

The time it takes to make expanded storage blocks available is directly related to the number and speed of the DASDs. Under normal operation, given average paging block sets of 9 to 12 pages and a mix of blocked and nonblocked paging, an IBM 3380 DASD can reasonably support around 130 pages per second. During the bursts of I/O activity created by migration, the same device can support upwards of 300 pages per second. The increase in capacity is possible because almost all of the I/O activity is blocked, and rotational and seek delays are minimized by using a DASD allocation scheme (moving wave) that tends to select pages from adjacent tracks.

VM/XA maintains its own 2-gigabyte virtual address space to hold routines and data.

Another function of the migrator is to move expanded storage blocks to DASD when some part of expanded storage is attached to a virtual machine. In this case the target is not age but expanded storage block numbers. All blocks within the target attached range are migrated before the expanded storage is given to the virtual machine. The attached expanded storage is also cleared to binary zeros for security. Attaching 256 megabytes of expanded storage to a virtual machine takes about one minute to migrate and clear the area.

The design of VM/XA for the exploitation of expanded storage allows very large expanded storages to be supported with minimal real storage resources. The reuse of the page table entries to hold expanded storage block numbers, the employment of dense allocation bit maps, and the migration scanning design allow VM/XA SP to efficiently support processor complexes with expanded storage that is much larger than real storage.

Internal storage management

In addition to the management of guest real storage, VM/XA must manage its own virtual and real storage for internal use. This storage is used for the control blocks, data structures, and modules required to

support potentially vast populations of virtual machines.

The VM/XA system virtual address space. VM/XA maintains its own 2-gigabyte virtual address space to hold routines and data. As with pageable-mode guests, only a portion of this address space has to be resident in real storage when the control program is executing. This allows certain routines and data to be maintained in expanded storage or on DASD when they are not actively being executed or referenced.

VM operating systems such as VM/SP and Virtual Machine/High Performance Option (VM/HPO) do not use dynamic address translation (DAT) when the control program (CP) itself is running. Even though the interpretive-execution facility permits DAT usage for the host program, the VM/XA control program also runs with DAT off. The transition from the DAT-off state to DAT-on is accomplished automatically by the SIE instruction when the state description specifies pageable mode. In fact, two levels of address translation are provided so that guest programs themselves may run DAT-on. The interpretive-execution facility restores the DAT-off state when returning control to the host.

Despite the fact that the VM/XA control program runs in nontranslate mode, there is still a need for portions of the CP nucleus to be nonresident to limit host real storage requirements. Examples include infrequently run routines and storage used to support spooling and the system directory.

To satisfy these CP nonresident storage requirements, a SYSTEM virtual storage address space is implemented in VM/XA. Even though DAT is not used to reference this virtual storage, VM/XA uses the same segment and page table structure required by DAT. This address space is a full 2048 megabytes, requiring a segment table two pages in size. The page tables to address this large SYSTEM virtual storage area are built dynamically on the basis of reference. The resident and pageable portion of the CP nucleus (about 2 megabytes) are mapped to the equivalent SYSTEM virtual storage area. Most VM/XA systems will start at page zero, but if the installation has reserved a fixed area for preferred virtual machines, the mapping to SYSTEM virtual storage starts beyond that area.

Because DAT is not used, VM/XA must perform its own translation. The Load Real Address instruction can be used for this translation, but more commonly VM/XA does a complete "software" translation. This process involves using bits 1–11 (the segment index) of the target virtual address to index into the system segment table to locate the appropriate page table. Then bits 12–19 (page index) are used to index into that page table to locate the page table entry. If the page is not resident, the corresponding fields in the PGMBK are interrogated to determine what action is appropriate (e.g., PGIN from expanded storage).

The module linkage mechanisms in VM/XA permit calling to nonresident modules. Calling is accomplished by establishing an address marker that defines the extent of the resident nucleus. Calls to modules whose address is higher than the marker are nonresident. The linkage mechanism calls the page manager requesting that the SYSTEM virtual address be made resident. The real storage manager returns the resident real address of the page, and the linkage mechanism then completes its processing. A pageable module is never called without assuming that a loss of control (to do the paging I/O operations) is possible. The pageable module performs its own base register relocation and cannot itself contain address constants that reference locations within the module. The module must also be less than or equal to a page. It may call upon other modules (resident or pageable) without fear that the module itself will disappear. The pageable module is locked into host real storage until a return is made from the module. The VM/XA linkage mechanisms can thus efficiently support a complex resident and nonresident module linkage mechanism even in a multiprocessor environment using the SYSTEM virtual storage construct.

The use of this SYSTEM virtual storage for spool buffers and other data follows a similar construct. The difference is that this storage is dynamically allocated from the "unused" storage area of the SYSTEM virtual storage. An allocation module locates the "next" SYSTEM virtual storage page that is available. It is marked as allocated, and the SYSTEM virtual address is returned to the caller. When a module needs to address such storage, it calls the real storage manager, passing it the SYSTEM virtual address. RSM will ensure that the page is resident (performing page fault processing as required) and will return the host real address of the page.

As previously stated, the system address space is defined by the same structures as those used for the address spaces of pageable-mode guests, providing both a realized and a potential advantage. The realized advantage is that the reorder and steal functions

of the real storage manager can be (and are) used to "police" the system address space. The potential advantage is that future releases of VM/XA could be made to run DAT-on without requiring changes to the system address space. At that time, the ESA/370 architecture could be exploited to maintain multiple system virtual address spaces, all easily addressable by the host program. In the future, specific types of system data might be isolated into different address spaces to improve overall system reliability, availability, and serviceability (RAS).

Free storage is used for the host control blocks and data required to maintain the virtual machine environments.

VM/XA free-storage manager. Free storage is used for the host control blocks and data required to maintain the virtual machine environments. The allocation and deallocation of such storage can occur at tremendously high rates and must be performed efficiently to allow optimal system performance. The VM/XA design of free storage has been dramatically changed from prior VM systems in order to meet the requirements of IBM's largest processors.

Earlier versions of VM required that a fixed region of real storage be set aside for the allocation of free storage. The size of the storage was generally determined at the time the system was generated, and although the free storage region could be extended dynamically, it could not be done without paying a performance penalty. Margolin et al.[9] show that very early on, VM implementers realized that managing this storage as a single chain of various-sized elements required excessive CPU time for searching and merging. Subpool techniques were introduced that provided separate queues for elements of specific sizes. All elements continued to be originally allocated from the global free-storage chain. However, for elements of appropriate size, deallocation would return the element not to the global queue but rather to a subpool queue which contained elements only of that size. A subsequent request for the same-sized element would be satisfied from that subpool.

Bozman et al.[10] provide an excellent review of techniques employed in early releases of VM and of those that were introduced in VM/HPO Release 2. These techniques greatly improved CPU efficiency but did not address the problem of free storage extension, nor did they completely eliminate the need to search a single chained list of elements. This continued to be required when:

- Subpools were depleted
- Requests were made for nonsubpool sizes
- Subpool elements were periodically returned to the global queue ("garbage collection")
- Elements allocated from extended free storage were returned

When designing a free-storage manager for VM/XA SP, it was hoped that the shortcomings of previous designs could be overcome. Thus the following design goals were established:

1. The installation should not be required to predict the amount of free storage the system would need.
2. The management of blocks that are frequently allocated and quickly released should be optimized in terms of CPU efficiency and provide acceptable storage utilization.
3. The management of blocks that are infrequently allocated and held for long periods of time should be optimized in terms of storage utilization and provide acceptable CPU efficiency.
4. The management techniques should be consistent as the amount of free storage required expands and contracts.
5. Debugging aids and integrity checking should be provided.

The result must be a robust, multiprocessor design with linear load-dependent performance characteristics. The free-storage manager must work well in all supported VM/XA environments and processors. As the speed, number of processors, and amount of free storage required increases, the CPU time necessary to manage free storage must increase linearly rather than exponentially. This increase can only be accomplished by employing techniques that limit the number of free-storage elements visited for each free-storage request.

The final design chosen, after several design and performance prototypes, had the following features:

1. No system generation of storage is required. All free-storage frames are dynamically obtained.
2. All short-term storage requests are managed as subpools.
3. All long-term storage requests are managed as simple chains associated with particular virtual machines or with the system as a whole.
4. The amount of storage available for both short-term and long-term allocation dynamically expands and contracts with no change in the management techniques employed.
5. All control blocks have an identifier as well as information about which function obtained and released the storage.

No static free-storage region. Previous VM releases required that a single large contiguous piece of real storage be reserved for free-storage allocation. The size reserved was either specified when the system was generated or defaulted to a fixed fraction of available real storage. Although the free-storage area could dynamically extend, such extensions invariably led to degraded system performance. To avoid such degradation, installations defined free-storage areas that were large enough to meet peek system demands. However, because this storage could not be used for other purposes, this approach had its own negative impact on off-peak performance.

The new VM/XA design allocates all free storage dynamically. There is no requirement for the installation to guess how much is needed.

Subpool processing. All storage needed for short-term requests is managed in subpools. A short-term request is generally a request that is not part of a virtual machine configuration description nor part of any system-managed function that may exist between guest sessions such as control blocks used for I/O operations. Short-term blocks may survive as long as several seconds or even minutes, but the actuarial tables reveal that the average life expectancy of these blocks is measured in milliseconds.

The sizes chosen for the subpools were derived after performing several system measurements which revealed the sizes of the most frequently requested control blocks. The design could have chosen one subpool for every possible valid request, but this was considered wasteful, particularly with the larger block sizes.

A valid free-storage request can be from 1 to 509 double words. With three double words added for control block identification and debugging, the actual size required is 4 to 512 double words (one page). All requests are rounded up to the next subpool size that will satisfy the request. The following subpool sizes and groupings were chosen:

1. 4 to 16 double words in 1 double-word increment—13 subpools
2. 18 to 64 double words in 2 double-word increments—24 subpools
3. 68 to 128 double words in 4 double-word increments—16 subpools
4. 144 to 256 double words in 16 double-word increments—8 subpools
5. 257 to 512 double words as 512 double words—1 subpool

The subpool sizes are not mixed within a free-storage frame. When a request for a certain size is made, the subpool anchor is examined. If there are no blocks of the requested size available, a new frame is obtained from the available list and is divided into storage blocks of the requested size. Any remainder is discarded. The entire chain of blocks is then placed on the subpool, allowing the initial request to be satisfied. This action has the advantage of "priming" the subpools so that subsequent requests can be immediately satisfied from the subpool.

The subpool blocks are chained in a LIFO (last-in-first-out) order. When a subpool block is returned, it is placed at the start of the chain and will be the next one allocated. This placement provides for cache efficiencies, making it more likely that a processor can reallocate a block that has not yet been discarded from its cache.

With the exception of the 4 to 16 double-word subpool sizes, internal fragmentation is possible because of rounding to the next higher subpool size. The potential for fragmentation is greatest for requests larger than 256 double words where a full frame is used to satisfy the request. Use of a full frame may appear to be very wasteful, but because these requests are short-term, the block is likely to be returned in milliseconds. Also, the grouping and rounding were chosen so that the most frequently allocated control blocks are from the smaller sizes where there is little or no fragmentation. Very large blocks are infrequently allocated.

This design largely avoids external fragmentation. Such fragmentation occurs only when a 4K frame

cannot be evenly divided into equal-sized elements, and the remainder is discarded. The subpool sizes were selected to minimize the amount of such waste.

A greater problem is that a request for a specific-sized element can only be satisfied from its corresponding subpool. If that subpool is empty, a frame will be taken from the available list even though there may be enough storage on other subpool queues to satisfy the request. To minimize this problem the subpools are periodically culled, and empty frames are returned to the available list.

All storage needed for short-term requests is managed in subpools.

Since the subpool sizes are used for frequently requested free-storage blocks, the design needed a low-level locking structure to allow for a high degree of concurrent access from multiple processors. Locking is controlled at the subpool level so that multiple processors may have concurrent access to different subpools. Observations of the subpool locks show very low contention because of simultaneous requests for the same size.

Long-term storage management. Long-term storage is used for control blocks that form part of a virtual machine definition or part of system data control. The former persist for the duration of the defined virtual machine (for example, from LOGON to LOGOFF of a CMS guest). The latter may persist from one VM/XA initial program load (IPL) to another (for example, a PROFS note, in the form of a spool file, will usually exist until read by the addressee).

Storage requests that are expected to be used for an extended period of time are not managed as subpools. The primary requirement for long-term storage is efficient storage utilization. Since the requests are infrequent, CPU efficiency is only of secondary importance.

The VMDBK is used by several control program and hardware functions, including the SIE instruction and dynamic address translation. About half of the full 4K frame allocated for the VMDBK is undefined. This area is used as a "private" free-storage area to satisfy requests for long-term storage that are directly associated with the guest. However, this private area is rarely large enough to hold all of the control blocks required for a typical virtual machine, and so the area needs to be extensible.

If a control block cannot be obtained from the private area, a global chain of guest free storage is examined to satisfy the request. A two-level structure is used. The first level is a chain of frame table entries (FRMTEs) which represent frames allocated for long-term requests. Within each frame, anchored in the FRMTE, is a chain of available storage blocks. The available storage blocks within a frame are sorted by size—smallest at the front and largest at the end. Initially a frame will contain one available block that is 512 double words long. The FRMTE also contains the size of the largest block in the frame.

A request for long-term storage is satisfied using a two-level first-fit algorithm. First, FRMTEs are examined (starting at the global-chain anchor) until one is found containing a block large enough to satisfy the request. Then the storage chain in that frame is examined until the first suitable element is found.

Once a suitable piece of storage has been found, it is unchained from the other blocks in the frame. If it is an exact match, it is allocated to the caller. If it is larger than requested, it is divided into the size requested and a remainder. The remainder is rechained into the available blocks in the frame sorted by size. An exception to this rule is applied to avoid excessive fragmentation. If the remaining piece is smaller than a "useful" size (less than eight double words), the large piece is not divided. The next larger block on the chain is examined. This next block may be divided into two pieces, leaving an acceptable size. If it is the last block in the frame, it will be divided regardless of the size of the remainder.

As pieces are given out from the frame, counters are maintained in the FRMTE showing how many double words remain in the frame. Once this number falls below a useful level, the FRMTE is removed from the high-level chain to avoid excessive chaining through FRMTEs, which do not have a size that can satisfy a request.

This method of examining the FRMTE for a requested size is used in the FRMTE that describes the VMDBK as well. Once this VMDBK frame is fully allocated, the global queue is examined. An alternative method was investigated. Since long-term guest control blocks are returned when the virtual machine is deleted, it made sense to keep guest requests separate rather than have a global queue for all guests. Thus each guest would have its own chain of FRMTEs with available storage queued in each frame, guaranteeing that all frames would be returned at guest LOGOFF. However, it implied that, on average, half of a frame of storage would be available and unused for each virtual machine. On benchmark systems of 5000 guests, this method consumed almost 10 megabytes of resident real storage; therefore this alternative was discarded.

System long-term storage is managed identically to guest long-term storage except that no VMDBK is used. A two-level chain of FRMTEs is handled exactly as described above.

Since the request rate for long-term storage is much lower than it is for short-term requests, a single global lock for the allocation of guest storage is used. There is a separate lock for the system-managed long-term storage queue. Performance measurements have shown that the spin time on these locks is negligible, indicating little or no contention.

As with the subpool storage method, the queue of available storage is initially empty. If the queue is empty or if no FRMTE has a piece large enough to satisfy a request, a new frame is obtained from the real storage manager. This frame is initialized to a single available piece, and the FRMTE is placed on the appropriate global queue.

Returning any long-term storage to the chain involves special processing to limit the amount of fragmentation and to keep the storage utilization at a high level. When long-term storage is returned, the FRMTE containing the piece is derived from the block address. The returned element is then merged with any adjacent unallocated elements in the same frame. The resulting piece will be sorted in the chain of blocks in the frame. If all storage has been returned, the frame is immediately returned to the available list.

Other free-storage systems in VM and in other operating systems have used or still use merging techniques to manage large strings of storage. As processor speeds and storage sizes increase, both the rate of free-storage requests and the number of elements examined to satisfy each request increase, leading to exponential increases in system overhead. In the VM/XA design, merging is limited to the available string of blocks within one frame. It effectively limits the amount of storage that needs to be examined to complete a merge operation. VM/XA does not use control blocks larger than a page, and no control block crosses a page boundary. These control-block limits make merging strings of available storage an effective nonsubpool storage management method.

Dynamic free-storage size. The previous sections showed that all frames used for free storage are obtained dynamically from the system available frame queue. It is done for short-term subpool storage frames and for long-term nonsubpool storage frames. One of the design interfaces for the free-storage manager specifies that a function may call for a block of storage and be assured that a block will be returned without deferring the requesting task. This specification permits tasks to use processor-specific storage (rather than task-specific storage) with the assurance that such storage cannot be destroyed by an intervening task. To allow this, the free-storage manager must be able to obtain a real frame without the necessity of deferring the requestor.

To obtain a frame, the free-storage manager examines the available list. If a frame is available, it is removed and used to satisfy the request. If no frame is immediately available, the steal task must be running on another processor to replenish the available list. Rather than waiting for steal to complete, the free-storage manager itself examines the FRMTEs representing the dynamic paging area, looking for an unchanged, unfixed frame. If such a frame is located, it is removed from the owner's queue (the page table entry is invalidated), and the frame is returned for free-storage use.

If no such frame can be located, the free-storage manager uses a reserved frame. Each processor in the VM/XA configuration reserves two frames at initialization for free-storage use. When a reserved frame is used, the steal task will ensure that the next frame made available to the system will be used to replenish the reserved frame pool. The system will not dispatch virtual machines but only handle system-scheduled work until the reserved frame pool is replenished. This limits the demands on free storage during this critical phase. If all reserved free-storage

frames from the pool are used and another request for storage is received, the system is abnormally terminated. Such terminations are rare and often caused by other failures that have essentially curtailed system operation.

When a free-storage frame becomes unused, it is a candidate to be returned to the available list. The return is made immediately for long-term storage frames since these requests are infrequent, and the chances of needing the frame again are low. For subpool storage a delay factor is introduced. Frames used in subpool storage are time-stamped. Periodically the system will examine these frames, and if the frame remains unused and has been for at least 15 seconds, it is returned to the available list. This timed delay for subpool frames is used because there is a higher probability that these frequently requested sizes may be needed within a short time span.

VM/XA free-storage management is designed to support very large numbers of guests, exploiting IBM's largest processors, such as the 3090 Model 600ES. Such support is accomplished by using subpool and nonsubpool storage management techniques that provide for efficient CPU and storage utilization. Observations have shown that the time spent managing free storage grows linearly with the system load. Peaks of 2000 requests per processor per second have been observed without adverse effects. Storage utilization efficiency (amount in use compared to amount allocated) generally has been in the range of 85 to 90 percent for both types of storage.

Summary

This paper has demonstrated how VM/XA SP is capable of supporting the System/370 Extended Architecture to its full potential, particularly in the area of virtual and real storage addressing and the exploitation of expanded storage. While VM/XA SP fully supports the use of the ESA/370 addressing extensions for guest use (i.e., for MVS/ESA), there is currently no direct exploitation of this feature for host control program purposes. However, the structure of the system lends itself to that possibility in future releases. Use of the ESA/370 addressing extensions by VM/XA could lead to further advances in the support of large numbers of users, particularly in areas where massive amounts of data are to be accessed and shared among users. The ability to use ESA/370 to reduce the cross-user data transfer protocols presently used could lead to significant performance improvements in these areas.

Cited references and notes

1. G. P. Bozman, "VM/XA SP2 minidisk cache," *IBM Systems Journal* **28,** No. 1, 165–174 (1989, this issue).
2. P. H. Gum, "System/370 Extended Architecture: Facilities for virtual machines," *IBM Journal of Research and Development* **27,** No. 6, 530–544 (November 1983).
3. Machines with the Processor Resource/Systems Manager™ (PR/SM™) feature can support up to six preferred guests, whereas machines without this feature are limited to a single preferred guest. (Processor Resource/Systems Manager and PR/SM are trademarks of International Business Machines Corporation.)
4. *IBM Virtual Machine/Extended Architecture System Product General Information,* GC23-0362, IBM Corporation; available through IBM branch offices.
5. The DIAGNOSE instruction, which otherwise has no meaning in a virtual machine environment, is used to request host services in much the same way as a Supervisor Call (SVC) instruction is used for other operating systems.
6. S. G. Tucker, "The IBM 3090 system: An overview," *IBM Systems Journal* **25,** No. 1, 4–19 (1986).
7. If the migrator finds that it is selecting an insufficient number of blocks per guest, it can adjust this target and thereby select "younger" expanded storage blocks.
8. M. G. Kienzle, J. A. Garay, and W. H. Tetzlaff, "Analysis of page-reference strings of an interactive system," *IBM Journal of Research and Development* **32,** No. 4, 523–535 (July 1988).
9. B. H. Margolin, R. P. Parmlee, and M. Schatzoff, "Analysis of free-storage algorithms," *IBM Systems Journal* **10,** No. 4, 283–304 (1971).
10. G. Bozman, W. Buco, T. P. Daly, and W. H. Tetzlaff, "Analysis of free-storage algorithms—revisited," *IBM Systems Journal* **23,** No. 1, 44–64 (1984).

General references

IBM Virtual Machine/Extended Architecture System Product CP Diagnosis Reference, LY27-8054-0, IBM Corporation; available through IBM branch offices.

IBM System/370 Extended Architecture Interpretive Execution, SA22-7095-1, IBM Corporation; available through IBM branch offices.

IBM Enterprise Systems Architecture/370 Principles of Operation, SA22-7200-0, IBM Corporation; available through IBM branch offices.

Geoff O. Blandy *IBM Data Systems Division, P.O. Box 100, Kingston, New York 12401.* Mr. Blandy is a senior programmer in the Advanced Technology department of the VM products organization in Kingston. He joined IBM in 1982, after several years as an MVS and VM systems programmer at Aetna Life and Casualty Company in Hartford, CT. He graduated from Wesleyan University in 1973 with a B.A. degree in psychology. He has concentrated on large VM system performance since joining IBM and is responsible for a number of modifications to VM/SP HPO and VM/XA. Among these are Active Wait, introduced in VM/SP HPO 3.4, for which he received the first software patent issued to

IBM Kingston. In 1985 he received an Outstanding Technical Achievement award for work done to improve the performance of HPO on dyadic processors. Mr. Blandy participated in the design and development of expanded storage and block paging support in VM/XA SF 2; minidisk-caching, virtual directory support, and "steal task" improvements in VM/XA SP 2; and architectural enhancements to the interpretive-execution facility.

S. Richard Newson *IBM Data Systems Division, P.O. Box 100, Kingston, New York 12401.* Mr. Newson is a senior programmer in the Advanced Technology department of the VM products organization in Kingston, currently on assignment to IBM Switzerland where he is supporting the supercomputing effort at CERN. He joined IBM Canada as a systems engineer in 1962 after graduation from the University of Alberta (Canada) with a B.Sc. degree in electrical engineering. He participated in the development and early support of TSS/360 and the installation of CP-67/CMS, then joined the IBM Cambridge Scientific Center in 1969 to work on the development of CP-67. In 1970 he was appointed development manager for the first version of the control program in VM/370. His focus on VM continued during the 1970s, with assignments in VM advanced technology, marketing support for VM/CMS in Brussels, Belgium, and design and development of the first VM/XA product. In 1983 Mr. Newson received an Invention Achievement Award in recognition of a patent filing for Multi System Mapping. This technology was later incorporated in the PR/SM feature of the IBM 3090 processors. After a staff appointment to the VM director, he rejoined the Advanced Technology department in 1985, where he participated in the design and development of expanded storage and block paging support in VM/XA SF 2; and minidisk-caching, virtual directory support, and "steal task" improvements in VM/XA SP 2.

Reprint Order No. G321-5354.