VM/XA SP2 minidisk cache

by G. P. Bozman

Given the growing disparity between CPU power and the speed of secondary storage, a data cache exploiting large processor storage has the potential to improve response time dramatically in many situations. The VM/XA SP2 minidisk cache facility, the result of research activity on the characteristics of interactive file-system activity, uses expanded storage to cache input/output to minidisks on the Conversational Monitor System. The size of the cache is dynamically adjusted by an arbitration process to optimize system performance. Several other functions improve the performance of the cache during periods of unusual I/O loads.

Caching is a widely used technique to improve performance in the presence of a memory hierarchy.[1,2] A cache is a relatively small, high-speed memory used to store the recently (or frequently) referenced contents of a slower, larger, and less-expensive memory in anticipation that these contents will be rereferenced soon, with a consequently significant improvement in performance. A cache is intended to provide performance approaching that of high-speed memory at a cost predominately that of the slower memory. Most contemporary high-performance computers contain a processor cache which is used as a buffer for references to primary storage. Some operating systems, e.g., the UNIX® system,[3] have data caches that are used to buffer, in primary storage, references to secondary storage. These data caches typically keep a set of the most recently referenced disk blocks.

This paper describes a data cache that is a component of Virtual Machine/Extended Architecture System Product™ 2 (VM/XA SP2). It is called a minidisk[4] cache because it uses expanded storage[5] to cache data from Conversational Monitor System (CMS) minidisks that are formatted into 4096-byte blocks. This minidisk cache can provide a significant performance improvement in systems where response time is at least partially influenced by I/O time.

Being able to dynamically vary its size in response to contending demands for expanded storage is a distinctive aspect of the minidisk cache.[6] An arbiter determines, in terms of global performance, the appropriate allocation of expanded storage. Therefore, the cache is self-tuning and automatically adjusts to a wide variety of system configurations and environments. In addition, several other unusual facilities provide high performance during periods of transient I/O stress.

In this paper a file-system activity study which provided the impetus for developing the cache is first described. Then some details are provided on a simulation study that was used to design the minidisk cache. Finally, a description is given of the initial prototype and how it evolved into the product version.

CMS file-system study

The minidisk cache in VM/XA SP2 has its origins in a study done at the IBM T. J. Watson Research Center facility in Hawthorne, New York, on the dynamic aspects of CMS file-system activity.

Data for this study were gathered using CMON, a CMS monitor written by David N. Smith, that allows information to be collected about activities within CMS for a particular user. It collects trace information about specific events within the CMS machine being monitored. For this study, CMON was extended to provide information at each READ and WRITE call and other file-system operations such as ERASE, RE-

NAME, and STATE (which determines the existence of a file with a given identifier on a specified minidisk). CMON was selected over a lower-level data-gathering facility such as the control program (CP) monitor because we wanted to study the file-system activity patterns at the logical, as opposed to physical, level.

CMON is invoked at log-on time, intercepts CMS interrupts and service-routine calls, and gathers pertinent data from CMS control blocks and work areas. A CP diagnose[8] handler was added so that CMON could uniquely identify each minidisk by volume serial identifier (VOLSER) and start cylinder. CMON collected raw activity data in a print spool file which was sent to a virtual machine and later reduced. One day's activity was collected from each user.

Although CMON was normally transparent to the CMS user, it was necessary for security and ethical reasons to request permission to monitor each of the randomly selected users. Because of this necessity, a considerable amount of time was spent setting up each monitoring session, and it was practical to monitor only 20 to 50 users on a large virtual machine (VM) system.

The study at Hawthorne was later performed on other internal IBM VM systems at Kingston and Poughkeepsie. All of the systems studied showed the following major CMS file-system activity characteristics:

- Activity was "bursty"—a user's command typically involved many file-system events and many files.
- There was a temporal locality of reference within a "burst"—i.e., files were often reaccessed within a very short time.
- Most files were read in their entirety and sequentially.
- Most file I/O events involved small files, but the relatively infrequent I/O activity of large files accounted for most of the bytes transferred.
- A few users accounted for a disproportionate share of the file activity. (This characteristic was corroborated by a sample of daily accounting data which showed that 1.3 percent of users performed 30 percent of the I/O events.)
- Byte read/write ratios ranged from 4.6 to 8.9 on the three systems.
- Approximately 50 percent of the files read were read more than once, and about 90 percent of all read requests were done to this set of files.

Cache simulation

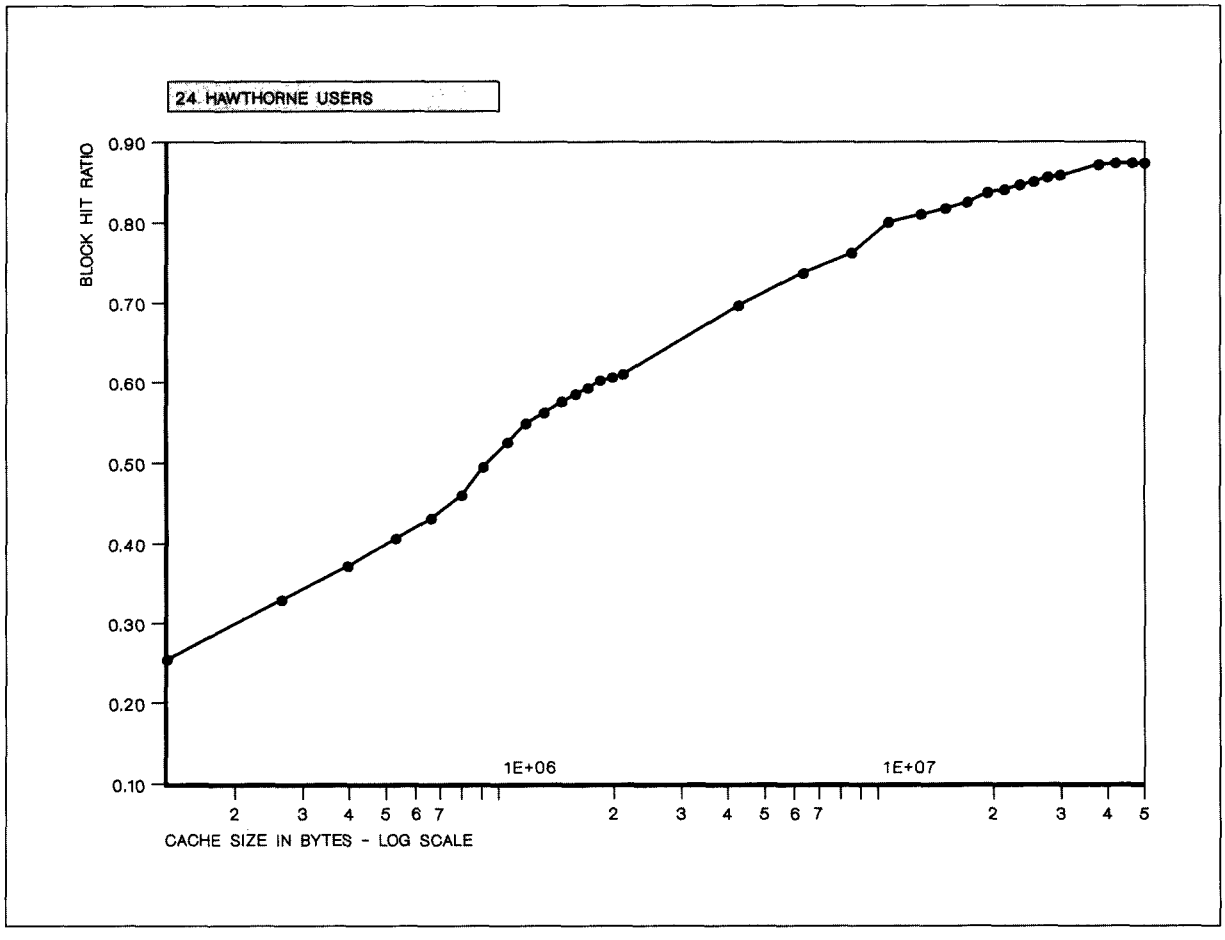
Since several of the file-system activity characteristics suggested that a data cache would improve CMS file-system performance, a cache simulator was written to experiment with the activity traces that had been derived from the CMON data. The traces for all of the users were combined by sorting them in order of event start time. Since there were a limited number of traces, we were concerned that they might not intersperse well and that there would be a long *run* of just one user's activity followed by a long run of another user's activity, etc.[9] Such noninterspersed activity would tend to impair the synergic cache effect that results when many users share files and get cache hits because some other user recently read the same file. This synergic effect would be expected on a large mainframe with many users. However, investigation showed that there was a high degree of interspersion of the activity. For example, at Hawthorne the average size of a trace for a single user was 2643 events, and after the combination of all traces, the average single-user run was 16 events, which was in the range of a typical "burst." Therefore, although the total number of users studied was more typical of a small VM/CMS system, the results of this study suggested that having a data cache on a large mainframe was beneficial.

Private virtual caches require significantly more space than a common cache to achieve the same hit ratio.

All of the cache simulation experiments had the following in common:

- Each simulation run started with an empty cache.
- The initial cache size was 64 kilobytes and was incremented for each simulation run.
- A 4096-byte minidisk block size was used for all runs.
- Cache blocks were replaced by a global least recently used (LRU) strategy.

Figure 1 Hit block ratio as a function of cache size



Since the cache was empty at the start of each run and there were a limited number of user traces, the simulation results tend to understate the hit ratios obtainable on a large mainframe with expanded storage.

Figure 1 shows the block hit ratio as a function of the cache size for 24 Hawthorne users sharing a common data cache. The block hit ratio is the number of (4096-byte) blocks found in the cache divided by the total number of blocks read. Even a relatively small cache provides a significant benefit; for example, with a 512-kilobyte cache the hit ratio is 0.41.

To show the benefit of having a common data cache as opposed to a private data cache in each virtual machine, each user trace was run independently, and the results were compared with a common cache by taking the mean hit ratio of all of the private caches and the sum of their sizes. Each private cache was incremented in size until it became large enough to contain all of the data referenced in the user's trace (i.e., became an infinite cache for that session). The cumulative size of private caches of size k for n users is $\sum_{i=1}^{n} \min(k, \textit{infinite cache size}_i)$. The corresponding mean block hit ratio is $\sum_{i=1}^{n} (b_i/s)h_i$ where b_i is the total number of blocks read by user i, s is the total number of blocks read by all users (i.e., $s = \sum_{j=1}^{n} b_j$), and h_i is the block hit ratio for user i at that cache size.

Figure 2 shows the results of this analysis. Clearly, private virtual caches require significantly more space than a common cache to achieve the same hit ratio. For example, with a 3-megabyte common cache the hit ratio is 0.67, but it requires 20 mega-

bytes of private cache to achieve similar results. Other research has demonstrated the space efficiency of a common centralized cache over distributed (nonprivate) caches such as those available in direct-access storage device (DASD) control units.[2] Also, the centralized cache may have a significant time advantage, depending on where the distributed cache is located. For example, a cache using expanded storage is significantly faster than a cache in a DASD control unit.

One reason a common cache performs so much better is illustrated in Figure 3. This simulation experiment cached only the data from the set of minidisks that were shared among two or more of the users. In addition to the hit ratio, this figure shows the fraction of the blocks hit because of data having been brought into the cache by another user. This synergic effect of a common cache is absent in a private virtual cache.

Finally, aside from the greater space consumption and absence of synergic effect, a hit on a virtual cache may require a page-in operation from a device of the same performance characteristics as the device on which the file is stored. In this event, the cache hit will not have improved performance because an equivalent I/O operation must still be performed.

Since the subset of shared minidisks yielded a higher hit ratio than the subset of private minidisks, and since the files in the shared subset were of potential benefit to multiple users, it seemed like a good idea to "protect" the shared subset from the private one. Experiments were run with the cache partitioned such that all of the data from the shared subset were placed in one partition and all of the data from the private subset into the other. Each partition had its own LRU replacement stack, so they were disjoint. Different relative partition sizes were used, but the results were always inferior to a common unparti-

Figure 2 Size of common cache versus sum of individual sizes

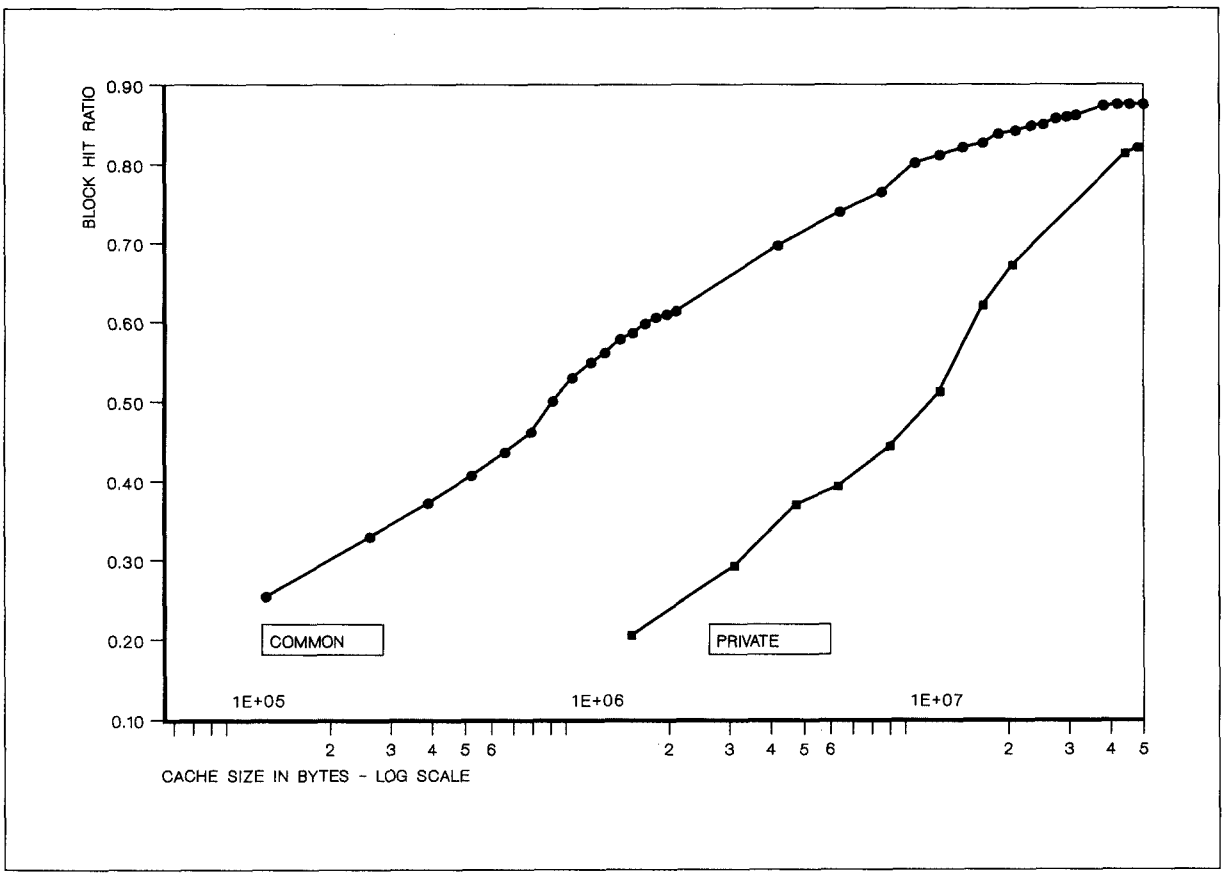
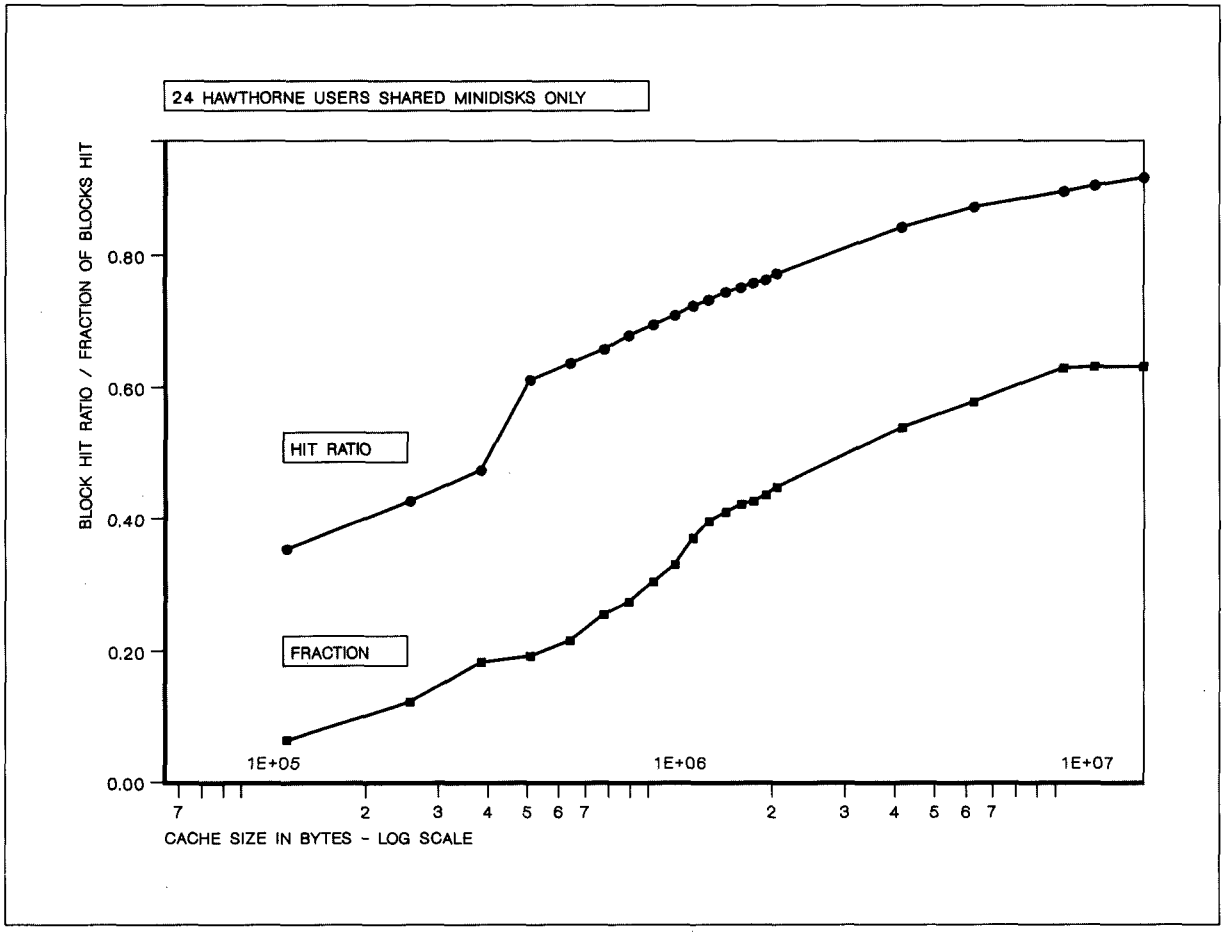


Figure 3 Block hit ratio and fraction of blocks hit because of reference by another user



tioned cache. This finding corroborates the results in Smith[2] since partitioning is equivalent to a distributed cache.

We also found that it was generally beneficial to cache all minidisks rather than to select those that have the highest degree of sharing.[10] This benefit is due to the temporal locality of reference within a "burst." Many of these rereferenced files are on private minidisks. Even a relatively small cache is able to contain the "burst" file references of the active set of users and consequently yield significant benefits.

The CMON study showed that most of the files read in CMS were small. For example, at Hawthorne, 45 percent of the files that were read were 2048 bytes or less in size. Storing these sparse 4096-byte file blocks in 4096-byte cache blocks results in a considerable amount of "wasted" cache space. The density of usable data in the cache could be increased by using a smaller granule of storage (i.e., cache line) in the cache. Using a cache line that was smaller than the file block size would increase the overhead and complexity of a cache but might be worthwhile if the space savings were significant.

Experiments were run with a cache line of 512 bytes, and the results were compared with a cache line of 4096 bytes (i.e., the file block size). The data density increased significantly (up to 50 percent at small cache sizes), but the improvement in hit ratio was much less than would have been expected by the increase in useful space. Further experiments showed that this disparity occurred because large files were

dominating the performance of the cache. A simulation was run excluding all files that were greater than 32 kilobytes in size. At Hawthorne 95 percent of the files were less than or equal to 32 kilobytes, but 60 percent of the bytes read were from files greater than this size. The hit-ratio difference increased significantly without the large files. For example, when all files were cached, the hit ratio for the 4096-byte cache line was only 5 percent less than the hit ratio for the 512-byte cache line with a 256-kilobyte cache, but when the large files were excluded, the difference was 18 percent.

Since all three systems on which we had taken CMON traces showed this dominance of large files, and since studies of other systems[11,12] suggested that this was a consistent characteristic of interactive file-system activity, it was decided that using a cache line smaller than the minidisk block size was not worth the increased overhead and complexity.

Prototype

A prototype minidisk cache was built and run in the Advanced Data Systems Laboratory in Hawthorne. This prototype was a centralized cache using global LRU replacement. The cache was a part of the control program (CP) of VM/SP HPO 4.2, and data were cached from minidisks that had a blocking factor of 4096 bytes. The data were cached in either expanded or primary storage (but not both). The time to copy a page from one page of primary storage to another is approximately the same as the time to *page in* a page from expanded to primary storage, so a hierarchical cache is not justified. There was significantly more performance leverage using expanded storage because of its greater availability and because there is less contention for it than primary storage.

The CP modules that service the CMS file system I/O activity (diagnose handlers) were modified to interrogate the cache manager on a read. If all of the blocks were in the cache, the blocks were copied to the CMS user's buffer, and the diagnose was synchronous. Cache misses went through the normal asynchronous I/O path. After cache misses completed, the blocks associated with them were inserted into the cache. Likewise, on the completion of output, the associated blocks were either inserted into the cache, or, if already present, updated. The option of a *write-in* cache[13] (as opposed to the *write-through* design used) was not practical because it would undermine the integrity of the CMS file system.

Dynamically adjusting the size of the cache in response to contending demands for the storage resource was the key characteristic of this prototype. A new module was added to CP which was an arbiter between demands for pages to back virtual memory and pages for use in the cache. The arbiter ran periodically and, based on an analysis of user wait-state samples, either adjusted the size of the cache, via an interaction with the cache manager, or left it unchanged. Heuristics were included in the arbiter to slow down or stop the incrementing of the cache when the performance benefit was slight or nil. Various tuning parameters in the arbiter allowed experimentation with trade-offs between responsiveness and dampening cache size oscillations.

A fair-share algorithm restricted the number of cache pages that any user could displace.

The arbiter approach was chosen over the alternative of letting the existing storage manager(s) select which pages should be cast out of processor storage. The reasons for this decision were:

- The arbiter allowed greater flexibility in managing processor storage. For example, the arbiter made it simple to bias in favor of either virtual memory pages or cache pages should this prove to be beneficial.
- Since most files are read sequentially and in their entirety, it is beneficial to "steal" cache blocks in file sequence. The global LRU replacement policy used by the minidisk cache has this attribute and consequently minimizes *partial hits* (i.e., some but not all of a list of blocks are in the cache). The LRU approximation algorithms such as *CLOCK*,[14] which are used by many storage managers, do not have this attribute.

The prototype had a *fair-share* algorithm which restricted the number of cache pages that any user could displace over a short interval. This algorithm

was designed not to constrain normal I/O activity but rather to limit the cache disturbance caused by pathological activities such as searching the contents of a minidisk for a specific string by reading every file in its entirety.

The fair-share algorithm worked as follows. For each interval of a minute, the cache manager counted the number of different users that had inserted blocks into the cache. Users who had only read data from the cache were not counted. When the minute interval expired, the fair-share limit for the next interval was calculated as the larger of *(cache_size)/(number_of_inserting_users)* or a lower bound. The lower bound was a heuristic value that always allowed a "reasonable" amount of cache insertion activity to occur per user. This lower bound adjusted the simple fair-share calculation to accommodate the known CMS file-system activity characteristics of strong temporal locality of reference and most I/O activities attributable to a small fraction of the users. The cache manager enforced the limit by counting the pages inserted by each user, and when the count exceeded the fair-share limit, no further pages were inserted. These counts were reset each minute. This *trailing* fair-share policy was not intended to impose a *true* fair share but rather to prevent unusual I/O activities from displacing a large fraction of the cache over a short interval of time.

The cache directory was implemented with a coalesced hashing algorithm.[15] This implementation allowed the directory to be fully loaded and still provide excellent performance. An adaptation of Bays' algorithm[16] was used to "rehash" the cache directory when required for either increasing the size of the cache or after a limit of empty directory space had been reached.

Product

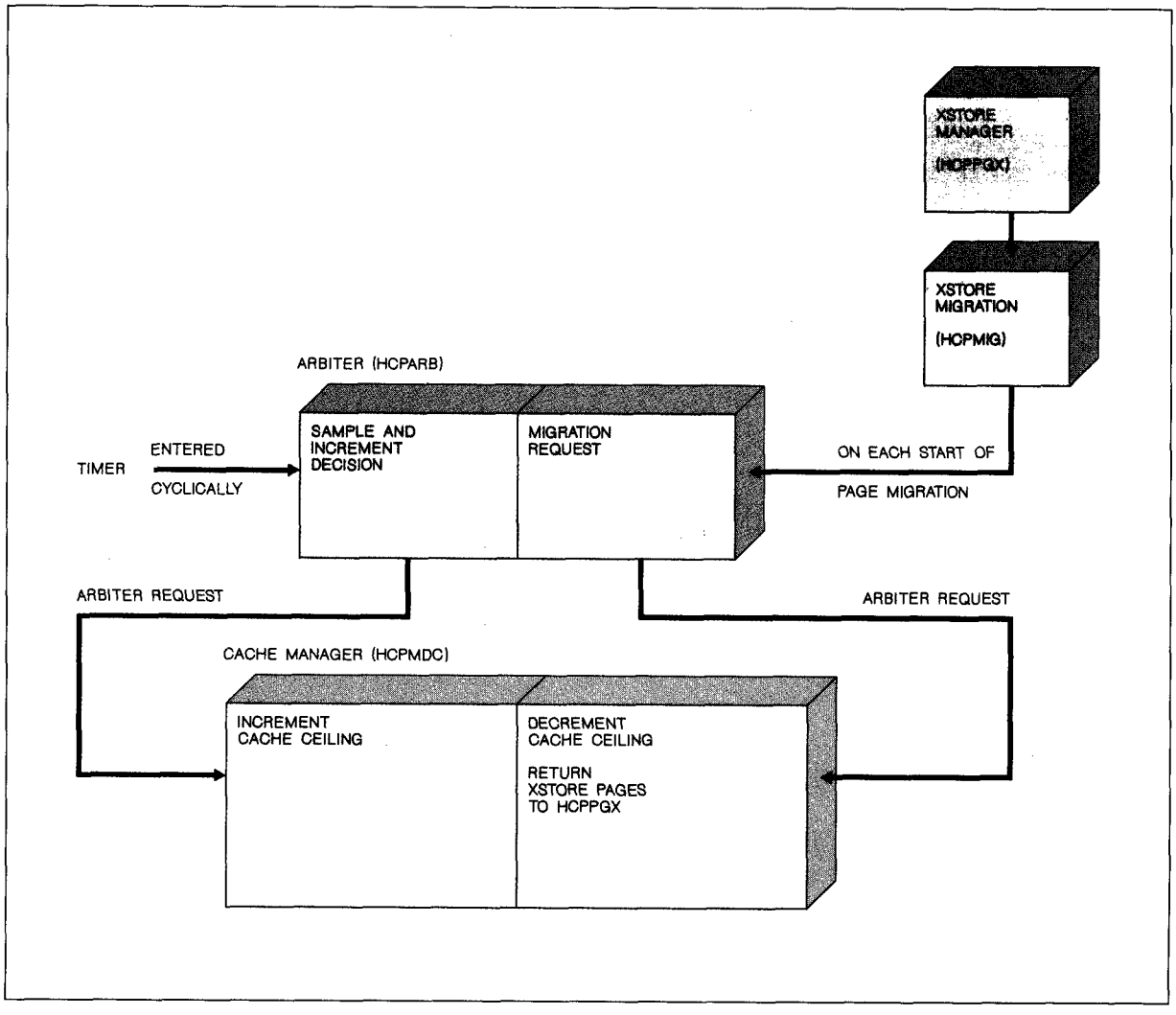
The prototype was converted to VM/XA in conjunction with the Advanced Technology Department of the IBM Kingston Programming Center. The product version uses expanded storage exclusively. The prototype structure was preserved, but several enhancements were introduced:

- The arbiter was modified to interface with the expanded storage migration routine so that a decision to decrease the size of the cache is made only upon invocation of the page migrator. The arbiter continues to dynamically make decisions to increment the size of the cache.
- Cache misses are inserted in the cache directory at the time of the miss. These directory entries are marked as *in transit*. Subsequent requests for the same block queue the user(s) on the cache directory entry until the block is inserted in the cache. Then all queued users are restarted at a point where they get a cache hit on the newly inserted block. Although this method is of limited value during normal operation, it greatly improves the VM recovery situation where hundreds of users are logging on after a system failure and reading the same blocks (e.g., accessing shared minidisks).
- I/O activity to CMS 4096-byte blocks that transfer less than the full block is cached. This allows the caching of CMS minidisk label I/O operations (80 bytes), the file directory root (64 bytes), and the pointer blocks of CMS transient modules (8 bytes). It is especially beneficial during VM recovery when CMS users are all reading the labels and file directory roots on the shared minidisks.
- When the cache has not yet reached its size ceiling (set by the arbiter), fair-share exclusions are inserted at the least recently used end of the LRU stack.[17] This action results in fewer fair-share exclusions during cache *growth* situations. For the typical condition where the cache has reached its size ceiling, the fair-share policy remains the same as in the prototype.
- A command was added to optionally set a minimum and/or maximum cache size. This command is intended for use in unforeseen situations where human intervention might beneficially limit cache-size oscillation. It might be useful, for example, when there is insufficient expanded storage to efficiently meet contending demands. When not specified by command, the minimum is zero and the maximum is the size of on-line expanded storage.
- Minidisks can be excluded from caching by including the minidisk option (MINIOPT) NOMDC on the MDISK record in the virtual machine directory.

The minidisk cache manager, HCPMDC, is called by the CMS I/O diagnose handlers for the new bimodal CMS (HCPDGB, HCPDGG[18]) and the traditional non-XA CMS (HCPDGD), and also by the module that services IUCV BLOCKIO[19] (HCPBIO) which is used by SQL/DS (Structured Query Language/Data System).

Blocks within a minidisk are invalidated from the cache when there is a virtual Start I/O/Start System Channel (SIO/SSCH) instruction to the minidisk (if it is writeable) or a CMS format of the minidisk. All blocks on a device are invalidated when the device

Figure 4 Relationship of page migration, arbiter, and minidisk cache manager



is detached from the system. Before a partition of expanded storage is attached to a guest virtual machine, all cached blocks within the partition are deleted from the cache.

As depicted in Figure 4, the arbiter, HCPARB, was modified to interface with the VM/XA page migration module, HCPMIG. The maximum size of the minidisk cache and its initial ceiling are set during CP initialization to be the size of on-line expanded storage and 16 megabytes, respectively. The maximum size and, if necessary, the ceiling are adjusted downward if a subset of expanded storage is attached to a guest virtual machine, or if the maximum has been set lower by command. At any time after the initial setting, HCPARB may raise the cache ceiling (up to its maximum value), if, during one of its cyclical invocations, its analysis of wait-state samples deems it to be beneficial. Until the page migration threshold is reached, both HCPMDC and HCPMIG compete for expanded storage pages via calls to HCPPGX, the expanded storage manager. When the migration threshold is reached, HCPMIG calls HCPARB before migrating pages from expanded storage to DASD. Based on its analysis of wait-state samples, the arbiter will either meet all, one half, or none of the migration goal. To meet all or part of the migration goal, HCPARB calls HCPMDC, specifying a cache ceiling that

is less than the current number of expanded storage pages in use for the cache. To conform to this lower ceiling, HCPMDC returns expanded storage pages to HCPPGX. After HCPARB returns to HCPMIG, a page migration occurs, if necessary, to acquire that portion of the migration goal not attained.

Minidisks are not cached if (1) the NOMDC option is specified on the MINIOPT directory statement, (2) the device on which the minidisk resides is dedicated to a virtual machine,[20] or (3) the device on which the minidisk resides is defined (in the I/O configuration module, HCPRIO) as being physically shared between processors. Otherwise all minidisks that have a blocking factor of 4096 bytes and have I/O operations done through the HCPDGB or HCPDGD diagnose handlers or the HCPBIO asynchronous block I/O handler are eligible for caching. Installations wishing to restrict caching should not rely on the blocking factor as it can be easily changed by the CMS user.

Minidisk cache statistics are included in VM/XA SP2 monitor output. The INDICATE LOAD command has been extended to include the minidisk cache hit ratio, minidisk cache blocks read per second from expanded storage, and minidisk cache blocks written per second to expanded storage. The QUERY XSTORE command displays the minimum, maximum, and current size of the minidisk cache.

Concluding remarks

Given the growing disparity between CPU power and the speed of secondary storage, a data cache exploiting large processor storage has the potential to dramatically improve response time in many situations. The self-tuning nature of the VM/XA SP2 minidisk cache allows it to respond robustly to a wide range of system configurations and environments. Also, the fair-share heuristic and cache directory queuing prevent otherwise pathological conditions from having a significant performance impact.

Acknowledgments

The minidisk cache product was developed by Jerry Bozman and Joann Ruvolo-Chong of the Computer Science Department of the IBM T. J. Watson Research Center facility in Hawthorne, New York, and George Bean, Geoff Blandy, and Dick Newson of the Advanced Technology Department of the IBM Kingston Programming Center in Kingston, New York. Without the assistance and support of the Kingston Advanced Technology, Design, Development, and Test organizations, the minidisk cache product would not have been realized. Hossein Ghannad and Ed Weinberger made key contributions to the CMS File System Study. Terry Donahue made a key contribution to the creation of a private virtual cache in CMS that was used to validate the cache simulation results and experiment with hashing techniques. Bill Tetzlaff fostered an environment in which this type of work could be accomplished. Finally, the referees were very helpful in improving the clarity of this paper.

Cited references and notes

1. A. J. Smith, "Cache memories," *Computing Surveys* **14,** No. 3, 473–530 (September 1982).
2. A. J. Smith, "Disk cache—miss ratio analysis and design considerations," *ACM Transactions on Computer Systems* **3,** No. 3, 161–203 (August 1985).
3. K. Thompson, "UNIX time-sharing system: UNIX implementation," *Bell System Technical Journal* **57,** No. 6, 1931–1946 (July–August 1978).
4. A minidisk is a contiguous subset of a real disk. In VM, a minidisk is used by the CMS file system to contain a set of files and their associated directory.
5. Expanded storage is page-addressable electronic storage. Data is moved between expanded storage and byte-addressable primary storage via two instructions, *page in* and *page out.*
6. After the minidisk cache was developed, we learned that a group at the University of California at Berkeley had independently developed a dynamic data cache for their Sprite Operating System.[7] The minidisk cache and the Sprite cache were developed for different machine and computing environments and differ in dynamic management technique and in other respects.
7. M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite network file system," *ACM Transactions on Computer Systems* **6,** No. 1, 134–154 (February 1988).
8. The System/370 *diagnose* instruction is used in the VM operating system as a stylized interface between virtual machines and CP.
9. A *run* is the number of consecutive file system events for a single user before an event for a different user occurred in the combined trace.
10. There are a few minidisks that are predominately write-only (e.g., a recovery log) that should be excluded, but these are rare.
11. J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A trace-driven analysis of the UNIX 4.2 BSD File System," *Proceedings of the Tenth Symposium on Operating Systems Principles* (December 1985), pp. 35–50.
12. R. A. Floyd, "Short-term file reference patterns in a UNIX environment," *Technical Report 177,* University of Rochester, Rochester, NY (March 1986).
13. With a *write-in* cache, data is placed in the cache without first having been written to secondary storage. There may be a performance advantage in periodically batching output (e.g., reduction of DASD *seeks*) and, also, in not having to perform

output for data that is subsequently overwritten or erased before being written to secondary storage. When using volatile storage for a write-in cache, there is a danger in certain types of system failures that the data cannot be written to secondary storage.

14. F. J. Corbato, *A Paging Experiment with the Multics System*, MIT Project MAC Report MAC-M-384 (May 1968).
15. J. S. Vitter, "Implementations for coalesced hashing," *Communications of the ACM* **25,** No. 12, 911–926 (December 1982).
16. C. Bays, "The reallocation of hash-coded tables," *Communications of the ACM* **16,** No. 1, 11–14 (January 1973).
17. That is, in position to be replaced when the cache ceiling is reached, unless they are rereferenced before then.
18. HCPDGG is used in bimodal CMS for I/O operations to the *short* blocks listed previously.
19. IUCV BLOCKIO is a facility that allows virtual machines to do asynchronous block input/output to minidisks.
20. Nondedicated minidisks that occupy an entire volume are cached. Minidisks can be exclusively accessed by a virtual machine without the use of the DEDICATE statement in the CP directory.

Gerald P. Bozman *IBM Research Division, T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Mr. Bozman is a programmer with a special interest in operating systems.

Reprint Order No. G321-5353.