I/O subsystem configurations for ESA: New roles for processor storage

by B. McNutt

I/O subsystem configurations are dictated by the storage and I/O requirements of the specific applications that use the disk hardware. Treating the latter requirement as a given, however, draws a boundary at the channel interface that is not well-suited to the capabilities of the Enterprise Systems Architecture (ESA). This architecture allows hardware expenditures in the I/O subsystem to be managed, while at the same time improving transaction response time and system throughput capability, by a strategy of processor buffering coupled with storage control cache. The key is to control the aggregate time per transaction spent waiting for physical disk motion. This paper investigates how to think about and accomplish such an objective. A case study, based on data collected at a large Multiple Virtual Storage installation, is used to investigate the potential types and amounts of memory use by individual files, both in storage control cache and in processor buffers. The mechanism of interaction between the two memory types is then examined and modeled so as to develop broad guidelines for how best to deploy an overall memory budget. These guidelines tend to contradict the usual metrics of storage control cache effectiveness, underscoring the need for an adjustment in pre-ESA paradigms.

The effective use of almost any computer system requires the juggling of three key resources: the central processor, the high-speed working memory provided in the processor (which may include both *central* and *expanded storage*), and disks for the permanent storage of files and databases. The processor memory and disk storage form a storage hierarchy, tied together by processor-initiated I/O operations.

On large systems, I/O operations are generally performed with the help of a *storage control* to access the disks and a *channel* to carry data between the storage control and the processor. A relatively recent addition to the storage hierarchy has been *storage control cache*, which is an extra working memory in the storage control that retains copies of the most recently used disk tracks.

The focus of this paper is the impact that Enterprise Systems Architecture (ESA) has had on the balanced use of the above resources in large systems. These include the computing environments provided by the IBM operating systems MVS/ESA* (Multiple Virtual Storage/Enterprise Systems Architecture), VM/ESA* (Virtual Machine/Enterprise Systems Architecture*), and VSE/ESA* (Virtual Storage Extended/Enterprise Systems Architecture). ESA provides a powerful range of capabilities for the use of processor memory as an I/O buffer area. Such an area allows I/O requests to be avoided by temporarily saving data previously obtained from disk storage. On the other hand, planning for

the I/O subsystem (the disk- and storage-control hardware) is often performed by taking as a given the I/O requirements of the applications that use the hardware. This "given" draws a boundary at the channel interface that is not well-suited to the capabilities of Enterprise Systems Architecture (ESA). Expenditures in the I/O subsystem can be managed, while at the same time improving transaction response time and system throughput capability, through a strategy of processor buffering coupled with storage control cache.

The key to accomplishing this is to control the aggregate time per transaction spent waiting for physical disk motion. The less time is spent waiting for disk motion, the fewer I/O subsystem resources such as actuators and storage controls are needed to service the stored data and the lower is the cost per megabyte of disk storage needed to achieve a given level of system throughput. This objective may at times conflict with and override the usual I/O subsystem objectives of minimizing I/O response time and maximizing storage control cache hit ratios.

This paper investigates how to think about and accomplish the game plan just outlined. Some of the questions addressed are:

- What types of data require which type of cache or processor memory?
- How can each type of data be given the amount of memory required?
- Should both cache and processor memories be used at the same time?
- If so, how much of each?

The following section begins by examining some actual patterns of potential ESA memory use to support specific files and databases. To explore the cache and/or processor memory requirements of individual files (or data sets, in MVS terminology) in a typical MVS environment, the *multiple workload* approach is used. A summary is given of the multiple workload approach and its application to memory planning.

The data show that a number of important files, such as database logs, do not use processor buffers and can be served at high speed only with storage control cache. Other files, such as small heavily used program libraries, can be served so well by processor memory that there is little need for storage control cache. For many types of data, including most large databases, both types of memory technology can be deployed to advantage at the same time. This is accomplished by using each technology in the following specific roles:

- Processor storage is used to hold individual page frames for long periods of time.
- Storage control cache contributes additional hits by staging the entire track of data following a requested record and holding it for relatively shorter times.

The succeeding section presents a study in more detail concerning the circumstances under which it makes sense to use both types of memory at the same time. Also discussed is the balancing of the amounts of the two types of memory against one another. For this purpose, a simplified statistical model, called *hierarchical reuse*, is used to gain a better understanding of the interaction between the two memory types. A summary of this model is also presented.

By applying the hierarchical reuse model, we show that, for most data, we should expect a balanced deployment of the overall memory budget, using both memory technologies, to be the most cost-effective strategy for achieving high performance. Guidelines for how to accomplish this are also developed.

We conclude by pulling together the results of the previous sections and discussing in greater detail the shift in I/O subsystem paradigms that ESA appears to require.

Patterns of memory use

We now consider individual file requirements for storage control cache and processor buffers at a real installation. This is done by showing early results obtained in a live MVS environment using a set of tools and procedures called Storage Hierarchy Analysis by Review of Data Sets (SHARDS).

SHARDS has been developed to facilitate an exchange of information about I/O subsystem workloads. The plan is to offer to run SHARDS at a number of installations in return for their participation in a survey of information about the types of files being used, patterns of file activity, and the locality of file reference. Summary results of the

survey are to be published for use by anyone interested in disk performance and capacity planning. To protect the privacy of the participants, customer names, file names, volume labels, or other identifying information is to be held in confidence and is not to be published.

SHARDS produces a picture of cache and processor storage requirements by individual file, in such a manner that the specific files to be supported by each technology can be selected or excluded one at a time. This is made possible by the *multiple workload* approach to cache planning,[1–3] in which the service objective for each caching technology is stated in terms of cache residency time (of which there are two flavors). A brief summary of the multiple workload approach is given first, followed by an examination of some of the early SHARDS results.

The multiple workload approach. The multiple workload approach to identifying cache memory requirements takes advantage of an analogy that can be drawn between the elements of a least recently used (LRU) list and the cars crowded together on a one-lane highway between two towns. Just as all types of cars on the highway require about the same amount of time to travel between towns, all types of data in the LRU list take about the same amount of time to go from the top to the bottom of the list.

This observation can be used to analyze each file's cache requirements separately from the requirements of any other files that may share the cache. The key that makes this possible is to state a service objective for the time-in-cache. Such an objective captures the operating state of the planned cache insofar as it affects any individual file. Planning for each file can proceed independently of any others, based solely on the operating state (time-in-cache) of the cache as a whole.

Single-reference residency time. Two flavors of the time-in-cache service objective are of interest. The most theoretically appealing, for the reasons just given, is the time to go from the top to the bottom of the LRU list. This is called the *single-reference residency time* in Reference 1, because it is the time spent in the cache by objects that are referenced only once. Subsequent authors have also introduced the terms *demotion time*[4] and *holding time*.[5] Future authors are strongly urged to pick from among the above three terms rather than adopting any new ones.[6]

Unfortunately, the single-reference residency time cannot be used for planning purposes without having a cache simulation tool that provides an output showing it. Recent tools have begun to provide this information so as to facilitate multiple workload planning.[4,5]

Average residency time. Because such tools have not been available until recently, a useful alternative has been to state the service objective as an *average residency time*, which is the average time for all cache visits, whether single-reference or otherwise. The average residency time will not be precisely the same for all files in the cache, but in this method of defining the service objective we proceed by assuming, for purposes of simplicity, that it is the same. This approach to multiple workload planning continues to be of interest for two reasons:

- Errors in cache hit ratio obtained from this simplifying assumption tend to be negligible.[1]
- The average residency time can be backed out from cache hit ratios.

The ability to back out the average residency time comes from Little's law. In its general form, that law states that for any system (where *system* is very broadly defined),

$$(\text{population}) = \begin{pmatrix}\text{residency}\\ \text{time}\end{pmatrix} \times \begin{pmatrix}\text{arrival}\\ \text{rate}\end{pmatrix} \qquad (1)$$

for the averages of these three quantities.

Equation 1 can be applied directly to a system comprised of the population of objects that have been staged into a cache. Because the storage required by these objects is the following:

$$\begin{pmatrix}\text{cache}\\ \text{storage}\end{pmatrix} = \begin{pmatrix}\text{population}\\ \text{staged}\end{pmatrix} \times \begin{pmatrix}\text{storage claimed}\\ \text{by each staging}\\ \text{operation}\end{pmatrix}$$

we can use Equation 1 to substitute for population:

$$\begin{pmatrix}\text{cache}\\\text{storage}\end{pmatrix} = \begin{pmatrix}\text{residency}\\\text{time}\end{pmatrix} \times \begin{pmatrix}\text{request}\\\text{rate}\end{pmatrix} \times \begin{pmatrix}\text{fraction of}\\\text{requests that}\\\text{cause staging}\end{pmatrix} \times \begin{pmatrix}\text{storage claimed}\\\text{by each staging}\\\text{operation}\end{pmatrix} \quad (2)$$

Equation 2 gives a way to estimate the average residency time directly from cache hit ratio data. For this reason, any desired cache simulation tool can be used to implement the average-residency-time version of the multiple workload approach.

A case study. We are now ready to apply the multiple workload approach to a specific case study. The installation examined in the case study was a large MVS/ESA environment running primarily online database applications. More specifically, most applications were constructed using Data Language/1 (DL/I) to formulate data requests and the Customer Information and Control System (CICS) to facilitate terminal interactions with the database. Database I/O was mostly performed using two of the several access methods provided by MVS:

- Virtual Storage Access Method (VSAM)
- Overflow Sequential Access Method (OSAM)

Both access methods provide for automatic processor buffering of I/O requests, with the size of the buffer usually being equal to one 4096-byte page frame. The analysis presented here is based on data obtained in early 1992.

We begin the analysis of the case study data by defining service objectives for the single-reference residency time in both storage control cache and processor storage. The objective for processor storage should exceed the objective for storage control cache by a factor of at least several times, because this ensures that the two technologies are able to cooperate effectively with one another. As discussed earlier, the key to effective cooperation is a subdivision of roles:

- Processor storage holds individual page frames for long periods of time.
- Storage control cache holds entire tracks for short periods of time.

For the purpose of the analysis below, we adopt the single-reference residency time objective of 300 seconds and 30 seconds, respectively, for processor buffers and storage control cache. Based on experience so far, these appear to be reasonable, middle-of-the-road objectives.

Our examination of the installation of the case study is based on a trace of all channel I/O at the installation during 30 minutes of the morning peak. The busiest 30 data sets appearing in this trace are presented in Figure 1. For each data set, the processor storage and cache storage requirements needed to provide the single-reference residency times of 300 and 30 seconds, respectively, are shown. The figure also shows the percentage of traced I/O requests that would be served out of the processor buffers or out of the storage control cache, given these memory sizes.

Note that the use of channel I/O as a base for expressing the percentages shown represents a compromise. Conceptually, a more appealing alternative would be to base the analysis on a trace of the logical data requests made by the application, including those requests served in the processor without having to ask for data from disk. A key aim of SHARDS, however, has been to capture a global picture of all disk activity, including all potential users of processor buffering or storage control cache. A global picture of this kind was believed to be practical only relative to channel I/O.

Inasmuch as channel I/O is the base for reporting, application requests that are hits in the existing processor buffers do not show up in the trace of channel I/O and are not reflected in the projected percentages of I/O served by the processor. Only the additional I/Os that may be intercepted by adding more buffer storage are shown. The SHARDS estimate of the processor buffer memory requirement does, however, include the existing buffer storage, up to the limit implied by the single-reference residency time objective.

With the dynamic cache facility of MVS/ESA, the use of storage control cache can be turned on and off on a data set basis. Similarly, the system administrator can choose which databases should be supported by a given buffer pool. To take max-

Figure 1 Busiest 30 files (data sets) in the case study and recommendations for their use of cache memory and processor buffers

Data Set	Memory Types Used		Processor Buffering Technique	I/O Rate	Projected Memory (MB)		Projected Percent of Trace I/O		
	CU?	PR?			Cache	Buffers	Disk Write	Served by Cache	Buffers
CICS journal	Y	N	n/a	26.6	1.1	0.0	100.0	97.2	0.0
CICS journal	Y	N	n/a	10.1	0.4	0.0	100.0	98.0	0.0
CICS journal	Y	N	n/a	10.1	0.4	0.0	100.0	98.0	0.0
CICS journal	Y	N	n/a	10.1	0.4	0.0	100.0	98.0	0.0
CICS journal	Y	N	n/a	9.3	1.7	0.0	0.0	84.8	0.0
CICS journal	Y	N	n/a	9.3	0.3	0.0	100.0	97.9	0.0
CICS system load lib	N	Y	prog lib	7.4	0.0	1.2	0.0	0.0	97.7
CICS journal	Y	N	n/a	7.3	0.3	0.0	100.0	98.0	0.0
Security control file	Y	N	n/a	7.2	0.9	0.0	47.9	94.6	0.0
CICS journal	Y	N	n/a	7.0	0.3	0.0	100.0	98.0	0.0
CICS application load lib	N	Y	prog lib	6.4	0.0	7.3	0.0	0.0	48.3
CICS application VSAM work area	Y	Y	Hiperspace	4.8	0.8	1.4	17.7	17.7	71.3
CICS application VSAM work area	Y	Y	Hiperspace	4.0	0.8	1.3	18.9	18.0	68.3
CICS application VSAM work area	Y	Y	Hiperspace	3.9	0.7	1.3	18.0	18.2	68.8
CICS DL/I VSAM database	N	Y	Hiperspace	2.9	0.0	2.0	2.7	0.0	44.1
CICS DL/I VSAM database	Y	Y	Hiperspace	2.9	0.0	0.3	9.3	13.4	85.8
CICS DL/I VSAM database	Y	Y	Hiperspace	2.9	1.3	1.6	11.5	15.7	49.3
CICS DL/I OSAM database	Y	Y	above line	2.2	1.1	1.3	17.3	21.0	39.8
CICS DL/I OSAM database	Y	Y	above line	2.2	0.6	2.0	9.4	21.9	57.6
CICS DL/I OSAM database	Y	Y	above line	2.2	0.4	1.8	13.2	26.1	57.6
CICS DL/I OSAM database	Y	Y	above line	2.2	0.6	2.1	10.8	23.6	55.0
CICS monitoring data (VSAM)	Y	N	Hiperspace	2.0	0.7	0.0	53.0	79.2	0.0
CICS DL/I OSAM database	Y	Y	above line	2.0	0.6	1.9	9.9	22.3	54.9
CICS application VSAM flat file	Y	N	Hiperspace	1.9	0.8	0.0	72.6	69.8	0.0
CICS DL/I OSAM database	Y	Y	above line	1.9	0.5	1.8	10.0	22.1	57.5
CICS DL/I OSAM database	Y	Y	above line	1.9	0.5	1.9	8.7	22.7	55.8
CICS DBRC recon file (VSAM)	Y	Y	Hiperspace	1.7	0.1	0.3	16.4	19.1	74.8
CICS DL/I OSAM database	Y	Y	above line	1.7	0.4	1.5	11.5	22.5	56.5
CICS DL/I OSAM database	Y	Y	above line	1.6	0.4	1.5	12.3	24.0	53.8
CICS application VSAM flat file	Y	N	Hiperspace	1.6	0.2	0.0	86.6	89.4	0.0
				...					
Summaries for:			Hiperspace	103.7	22.2	35.7	22.1	25.4	56.3
			above line	61.6	19.4	45.2	8.6	17.7	56.5
			prog lib	16.5	0.4	8.6	0.0	6.6	70.2
			n/a	105.1	8.5	0.0	82.5	94.7	0.0

imum advantage of this flexibility, SHARDS uses a spreadsheet program so that the planner has the option, for each data set, of whether or not to allocate that data set's projected processor and storage control cache memory requirements. This choice is indicated by the columns CU? and PR? in Figure 1.

Figure 1 also indicates the method (if any) by which large buffer areas can be defined for use by each data set. Some of the methods indicated in the figure include:

- Hiperspace*—special expanded storage extensions of the VSAM Local Shared Resource buffer area
- Above line—large buffer areas defined by using areas of the virtual address space beyond the line represented by the largest 24-bit address. Above-line buffer areas, addressed by 31-bit addressing, use a combination of central and expanded storage under the control of the paging subsystem.
- Prog lib—special storage for frequently used programs, provided through the Virtual Lookaside Facility/Library Lookaside (VLF/LLA)

The figure does not contain examples of the full range of facilities available for using large processor buffer areas. (For readers particularly interested in MVS buffering, some not shown include PDSE [partitioned data set extended], CLIST [command list], and catalog support, VIO [virtual I/O], and the Hiperbatch* facility.)

Some data sets cannot make use of large processor buffer areas via any of the standard methods just outlined. Examples illustrated by Figure 1 include CICS journals (i.e., database logs) and the security control file, which are denoted by *n/a* in the column labeled Processor Buffering Technique. Also shown are three VSAM flat files (one used by the CICS monitor facility) for which large processor buffers were found to be ineffective owing to the sequential method of access. Figure 1, therefore, shows the deployment of storage control cache memory, but not the deployment of processor memory, for the logs, the security control file, and the three flat files.

In contrast, an approach based on processor buffers was adopted to support the program libraries (except for some of the less active ones not included in Figure 1). Although no attempt was made to do so in the case study, it is possible to obtain 100 percent hit ratios in the processor for a few selected program libraries, by setting aside enough storage to contain them entirely. Similar results are also possible for a few other highly specialized data sets. Enough processor storage can be set aside to contain entirely, however, at most a small fraction of all data sets. Disk storage (measured in gigabytes) has three orders of magnitude more capacity than does processor storage (measured in megabytes).

Most of the databases presented by Figure 1 can make effective use of both processor storage and storage control cache. The recommended amount of storage control cache, as shown in the figure, is usually smaller than the amount of processor storage. This reflects the specialized role of the cache memory, which is to capture the following:

- Write activity
- Those read hits that come from staging an entire track of data rather than a single record

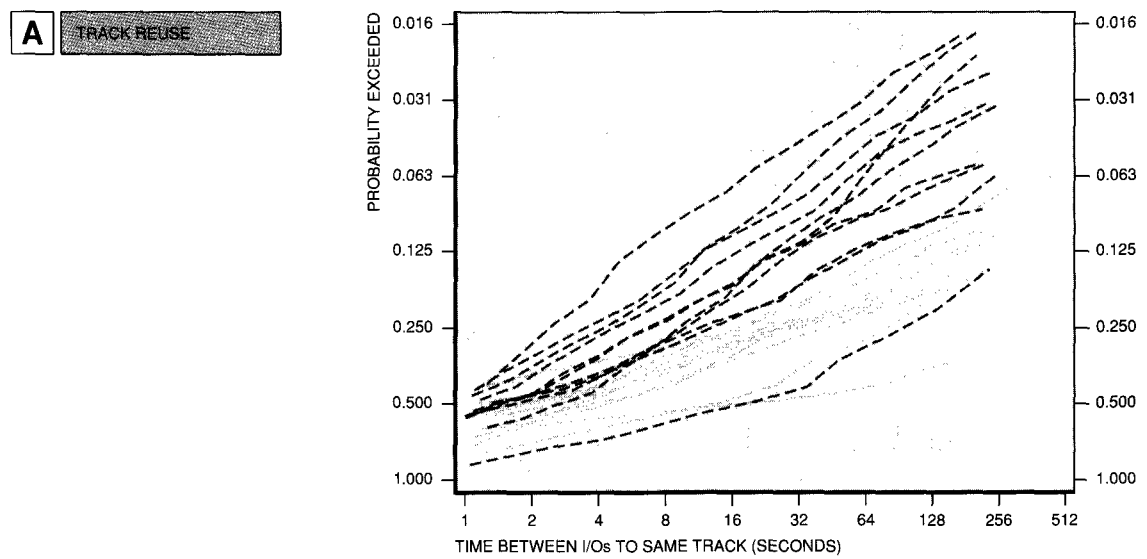
Because of their low percentages of read hits compared to overall reads, the databases presented by Figure 1 might appear to be making ineffective use of storage control cache, if judged by the read-hit-ratio measure of cache effectiveness. These data sets are nevertheless being supported in an efficient and cost-effective manner. The fact that this is not revealed by the traditional read-hit-ratio metric underscores the need for a shift in traditional planning paradigms.

Expectations for memory interaction

As previously discussed, time-in-cache objectives can be established for storage control cache and for processor buffers, so that the two types of memory work cooperatively. But the functions provided by the two memories partially overlap. Read hits in the processor cannot also be hits in storage control cache. Does it really make sense to use both types of memory at the same time on the same data?

We now address this issue by applying a simplified statistical treatment called the *hierarchical reuse* model.[7] Despite its simplicity, the hierarchical reuse model provides a highly serviceable description of realistic reference patterns. The results of the model show how the tradeoff of the two storage technologies is driven by the basic difference in their memory management strate-

Figure 2 Empirical checks of the hierarchical reuse model



gies. We find that for typical data there are two conclusions:

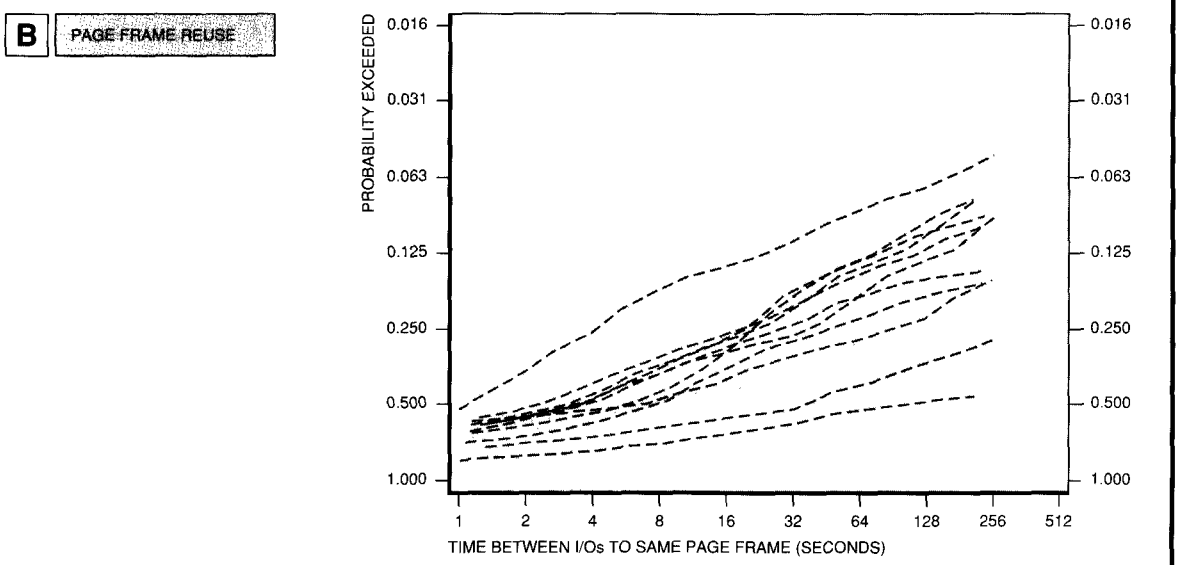
1. The best method of deploying a given memory budget is to use a relatively larger amount of processor storage and a small-to-moderate amount of storage control cache.
2. Within this guideline, overall performance is highly insensitive to the exact ratio of memory sizes.

The second conclusion is a big help in practical applications. SHARDS takes advantage of it by applying the same time-in-cache objectives outlined earlier in this paper to each type of memory, whether the two types are being used together or separately. This succeeds in achieving a sound balance of the two memories because it yields a result that is in the ballpark, that is, close enough given the second conclusion.

For simplicity in dealing with the fundamental issue of memory management in page frames versus tracks, we consider a reference pattern that consists of reads only. Also for simplicity, we use a model of storage control cache operation in which any reference to a track contained in the cache is considered to be a hit. The probability of a so-called "front-end miss" (request to an uncached portion of a cached track), normally very small, is assumed to be zero.

Hierarchical reuse model. The hierarchical reuse model of reference locality is motivated by the observation that the causes of data reuse tend to have a hierarchical structure. Thus, I/Os are requested by subroutines that are a part of main routines that are invoked by transactions that, in turn, are employed to accomplish some overall task. Because of this, the hierarchical reuse model hypothesizes that the probability structure of data reuse at long time scales should mirror that at short time scales, after the time scale itself is taken into account.

For example, consider two tracks in storage control cache: (1) A *short-term track*, last referenced 5 seconds ago, and (2) a *long-term track*, last referenced 20 seconds ago. The hierarchical reuse model predicts that the short-term track has the same probability of being referenced in the next



5 seconds as the long-term track does of being referenced in the next 20 seconds. Similarly, the short-term track has the same probability of being referenced in the next one minute as the long-term track does of being referenced in the next four minutes.

Suppose U represents the time from the last reference to the next re-reference of a given track (or page frame); and suppose that u_0 is an arbitrary interval of time (e.g., u_0 might represent the time intervals of 5 or 20 seconds used in the examples of the short-term and long-term tracks, respectively). Then a more formal statement of the hypothesis above is:

Hierarchical reuse hypothesis: The conditional distribution of the quantity

$$\frac{U}{u_0} \mid U > u_0$$

does not depend upon u_0. Moreover, this distribution is independent and identical across periods following different references.

A hypothesis of this form has to be constructed with some lower limit on the time scale u_0; otherwise, there is danger of dividing by zero. For storage control cache and processor memory, the lower limit appears to be much lower than any of the time scales of interest (some fraction of one second). The lower limit, although it exists, has little or no practical effect on the application of the model. For this reason we shall usually ignore it.

The behavior just predicted by the hierarchical reuse hypothesis is a special case of *statistical self-similarity*, a feature often seen in the study of fractals. Indeed, it is shown in Reference 8 that a random variable U that satisfies the conditions stated in the hierarchical reuse hypothesis must belong to the *hyperbolic* family of distributions; i.e., U can be characterized by

$$P[U > x] = ax^{-\theta} \tag{3}$$

for some constants a and θ, and for sufficiently large values of x (values above the lower limit mentioned earlier).

The thrust of Equation 3 is that if U is plotted versus x in a log-log plot, the result should be a straight line. Figure 2 presents a check of this hypothesis against trace data collected at 11 VM installations[2] in 1988. Every volume at each installation was traced. The volumes were then divided into data pools, depending on the type of data. Figure 2 presents the results for the pool of system data at each installation (shown as a dashed line) as well as the pool of user data (shown as a solid line). The left side presents a log-log plot of the distribution of U as defined in terms of references at the track level. The right side presents a similar plot of the distribution defined in terms of references at the page-frame level, where each page frame contains one VM disk record.

Figure 2 comes strikingly close to being the predicted collection of straight lines. This makes Equation 3 a highly serviceable approximation to real reference behavior. Little would be lost by replacing any of the curves of Figure 2 by a straight line, as specified by suitable choices of the parameters a and θ.

An important aspect of the data presented in Figure 2 is the difference in slopes between the left and the right side. For example, user data typically seems to exhibit a slope corresponding to a value $\theta \simeq 0.25$ in the plot of track reference locality. By contrast, user data in the plot of page-frame locality tends to exhibit about half of this slope, $\theta \simeq 0.125$.

The value of θ is a measure of cache *responsiveness* with respect to increasing time-in-cache. Some intuition about θ can be gained by considering again the idea of a hierarchical set of routines/programs/transactions/tasks that might generate repeat references to a given data object. In this framework, the value of θ is related to the average number of repeat references to a given object that are expected to occur at a given level of the hierarchy. That is, it is an indirect measure of the amount of branching that affects a given data item. This clarifies why the slopes on the left of Figure 2 are greater than the slopes on the right. The difference in slope reflects the fact that more repeat references can be expected to a given track than to a given record.

It can be shown that Equation 3 has a number of interesting consequences.[7] The miss ratio m, single-reference residency time τ, average residency time T, and memory size s, relate to one another in a very simple series of equations:

$$\left.\begin{aligned} m &= a\tau^{-\theta} \\ \tau &= (1-\theta)T \\ T &= \left(\frac{s}{zbr}\right)^{\frac{1}{1-\theta}} \end{aligned}\right\} \qquad (4)$$

where r represents I/O rate, z represents stage size (average amount of storage claimed in order to add a new entry into memory), and b is a constant related to a:

$$b = a(1-\theta)^{-\theta}$$

These equations can be applied to obtain the miss ratio as a function of cache size for either storage control cache or processor buffers operating as a separate technology. If the two technologies operate together, some hits occur only in the processor, which otherwise would have occurred in storage control cache. The effect of this on the miss ratio in storage control cache is easiest to see when the single-reference residency time in the processor is shorter than that in the cache, i.e., when $\tau_p \leq \tau_c$ where the subscripts p and c are used to denote processor and storage control cache memories, respectively. In this case, all the hits in the processor overlap with hits that would have occurred in the cache by itself, assuming that the cache's single-reference residency time is held fixed. The effect of processor buffering is, therefore, to reduce the number of requests to the cache without any reduction in cache misses. As a result,

$$m_c' = \frac{m_c}{m_p} \qquad (5)$$

where the prime indicates the miss ratio in the combined configuration.

A more useful configuration is one in which $\tau_p > \tau_c$. The analysis of this case is more complex, but it can be shown in this case also that the hit ratio in the cache when running in combination with processor buffers can be estimated as a function of the hit ratios for the two memory technologies[3] operating separately:

$$m_c' = \frac{m_c}{m_{p|\tau_p = \tau_c}} \tag{6}$$

where the suffix starting with "|" indicates that the processor miss ratio is evaluated at the single-reference residency time of the storage control cache.

When applied to Equation 6, the reasoning of Reference 7 produces a set of relationships closely analogous to those for a single memory. Provided that $\tau_p > \tau_c$, we have:

$$\left.\begin{aligned} m_c' &= \frac{a_c}{a_p}\,\tau_c^{-(\theta_c - \theta_p)} \\ \tau_c &= [1 - (\theta_c - \theta_p)]\,T_c' \\ T_c' &= \left(\frac{s_c}{z_c b_c' m_p r}\right)^{\frac{1}{1-(\theta_c - \theta_p)}} \end{aligned}\right\} \tag{7}$$

where

$$b_c' = \frac{a_c}{a_p}\,[1 - (\theta_c - \theta_p)]^{-(\theta_c - \theta_p)}$$

Balancing the memories. Equations 4, 5, and 7 provide a practical method to estimate the miss ratio as a function of memory size in a configuration that includes both storage control cache and processor memory. The delay D to serve a given I/O request can therefore be estimated as well:

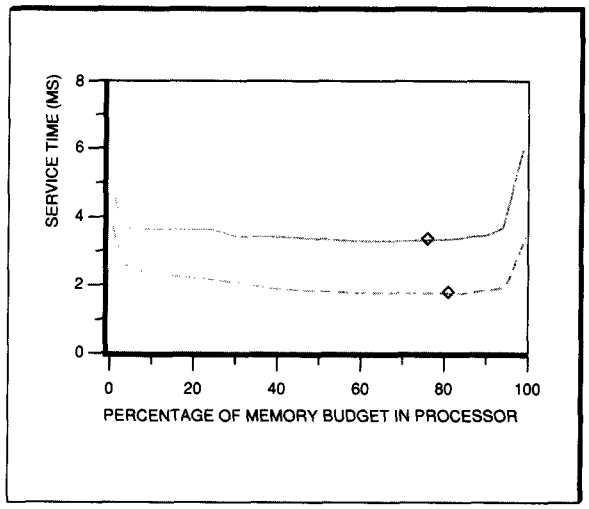
$$D = m_p D_p + m_p m_c D_c \tag{8}$$

where

D_p = increment of delay caused by a miss in the processor buffer (i.e., the time required to obtain the data from storage control cache)
D_c = additional increment of delay caused by a miss in the storage control cache (native device service time less time for cache service)

Figure 3 presents the result of applying Equation 8 across the range of memory sizes that yield a fixed total size of 0.25 megabytes per I/O per second. This figure uses average values of the user data pools (solid line) and system data pools (dashed line) presented in Figure 2 as the parameters determining cache behavior. The quantities

Figure 3 Tradeoff of the two memory types, where the diamonds mark 10-times ratios of the single-reference residency time



D_p and D_c are assumed to have the values 2.5 and 12.5 milliseconds, respectively (making total service time on the native device equal to 15 milliseconds). For the extreme case where either memory size is zero, the miss ratio is taken to be unity. To avoid the lower limit of the hierarchical reuse time scale, the regions involving single-reference residency times of less than one second for either memory are bridged using interpolation.

Figure 3 establishes the two conclusions promised earlier in this paper:

- The best method of deploying a given memory budget is by using a relatively larger amount of processor storage and a small-to-moderate amount of storage control cache.
- Within this guideline, overall performance is highly insensitive to the exact ratio of memory sizes.

From a practical standpoint, a good way to at least come close to the optimum memory balance is to plan for a time-in-cache service objective in the processor that is approximately a factor of ten greater than the same objective in storage control cache. Figure 3 shows the points where this occurs for the user and system cases.

Although the exact ratio of memory sizes is not a sensitive one, it is still interesting to ask where the

actual minimum in the service time occurs. For this purpose, it is useful to generalize slightly the treatment of Figure 3 by assuming that the total memory budget is given in dollars rather than in megabytes. If both types of memory are assumed to have the same cost per megabyte (which is roughly true at current prices), then this reduces to the framework of Figure 3.

Now suppose we wish to minimize the total delay D subject to a fixed budget

$$s_p E_p + s_c E_c = B \tag{9}$$

where

E_p = cost per megabyte of processor storage
E_c = cost per megabyte of cache storage

The Equations 7 can be solved to show that the minimum occurs when

$$\frac{s_c E_c}{s_p E_p} = \left(\frac{\theta_c}{\theta_p} - 1\right)(1 - \theta_p)\frac{1}{1+\delta} \tag{10}$$

where

$$\delta = \frac{D_p}{D_c}\frac{1}{m_c'}[1 - (\theta_c - \theta_p)]$$

Application of Equation 10 requires an iteration on the value of the cache miss ratio. The miss ratio can initially be set to an arbitrary value such as 0.5, then recomputed using Equations 10 and 7. Convergence is very rapid, so that three evaluations of Equation 10 are enough to obtain a precise result.

In the present context, however, we are not so much interested in performing calculations based on Equation 10 as in using it to gain insight. For this purpose, it is helpful to consider what happens if the goal is simply to minimize the number of requests to be served by the native device. In this case, we take into account only D_c and assume that D_p is zero. With this simplification, Equation 10 reduces to:

$$\frac{s_c E_c}{s_p E_p} = \left(\frac{\theta_c}{\theta_p} - 1\right)(1 - \theta_p) \tag{11}$$

This result shows clearly that the crucial determinant of the best balance between the two memories is the difference in their *cache responsiveness* (i.e., values of θ). As long as there is any tendency for references to different page frames to cluster into groups, thereby causing a greater amount of use of a given track than of a given page frame, then some amount of storage control cache is appropriate. The stronger this tendency grows, the greater the role of storage control cache becomes in the optimum balance. Using as an example the values for θ of 0.25 in storage control cache and 0.125 in processor memory (the values mentioned earlier in this paper for user data), Equation 11 indicates that the fewest native-device accesses occur when the ratio of the storage control and processor portions of the memory budget is

$$\left(\frac{0.25}{0.125} - 1\right)(1 - 0.125) = 0.875$$

This means that $1/(1 + 0.875) = 54$ percent of the total budget is allocated to the processor. If, instead, the values of θ are 0.35 in storage control cache and 0.225 in processor storage (typical values for the system data in Figure 2), we would allocate 70 percent of the total budget in the processor to get the fewest native device accesses.

As indicated by Equation 10, the memory balance that minimizes the total delay D involves a small upward adjustment in processor memory compared to the results just given. Assuming for simplicity that both memory costs E_c and E_p are equal, the fractions of the total storage needed in the processor to produce minimal delay are 62 and 78 percent in the user and system cases, respectively.

It is worthwhile to reiterate that achieving the optimum balance is not important in practice. As Figure 3 shows, what matters is to achieve some balance, so that the larger portion of the memory budget is in the processor and a small-to-moderate portion is in the storage control cache. This is sufficient to ensure that the delay per request is close to the minimum that can be achieved within the memory budget.

It should again be noted, as at the end of the section on patterns of memory use earlier in this paper, that in a configuration that displays the

desired balance of memories, the read hit ratio may well be below the often-recommended guideline of 70 percent. In the user and system configurations just discussed that yield the minimum delay D, the storage control cache hit ratios are 58 and 64 percent, respectively. The low hit ratios are mitigated, however, by the overall load reduction due to processor buffering. Average utilization of the I/O subsystem actuators and paths is substantially lower in the selected configurations than it would be without processor buffers.

Concluding remarks

We have used the multiple workload approach to gain insight into the manner in which both processor and storage control memory can be used to provide fast and cost-effective service for disk I/O. The application of the approach to a specific case study shows that it is helpful to divide the files on disk storage into three categories:

1. Files for which buffer memory is best provided in the storage control. Examples drawn from the case study include flat files as well as files, such as logs, that do not have the capability to use processor buffering.
2. Files for which buffer memory is best provided in the processor. Examples drawn from the case study include several program libraries (which can be contained entirely in processor memory if they are small enough) as well as one database that had almost no tendency for nearby records to be referenced together.
3. Files for which a mixed-buffering strategy is most effective. This group includes most of the databases examined in the case study.

For the data in the third category, the larger amount of memory should usually be provided in the processor with a small-to-moderate amount being provided in storage control cache.

Collectively, these conclusions demand that two key assumptions often made in the disk planning process must be abandoned. These assumptions are usually implicit, but at this stage it would be worthwhile to bring them into the open:

1. A system throughput requirement in transactions per second directly implies a corresponding I/O rate requirement in I/Os per second. The problem with this assumption is that the number of I/Os per transaction can be adjusted for any given transaction type by changing the number of processor buffers available for that transaction type. If desired, system throughput can be increased with no corresponding increase in I/O rate.
2. Transaction response time and system throughput capability are directly coupled to disk response time, and therefore to the hit ratio of the storage control cache. This assumption breaks down when the number of I/Os required by a given type of transaction is not fixed. Even if the response time per I/O increases, transaction response time might still go down, and system throughput capability may increase if the number of I/Os per transaction is reduced. This outcome can be expected to occur routinely in planning for MVS/ESA, VM/ESA, and VSE/ESA systems.

All is not lost, however. The following premise continues to stand up, and helps to act as a replacement for statements 1 and 2 above:

3. Transaction response time and system throughput capability are directly coupled to the aggregate time per transaction spent waiting for physical disk motion.

In Enterprise Systems Architecture (ESA), many capabilities are available to reduce the amount of such motion and to target the reductions to the types of transactions that are most response-time critical. These include processor buffering techniques provided by ESA, the use of dynamic cache management to target storage control cache at specific files, and tools to identify which files are most impacted by disk delays.[4,5] It is easy to waste time and effort, however, trying to solve the wrong problem, i.e., trying to minimize I/O subsystem response time when this is not the most helpful objective.

The capabilities of ESA can best be leveraged to achieve cost-effective I/O subsystems by keeping statement 3 above in mind. The less time is spent waiting for physical disk motion, the fewer I/O subsystem resources such as actuators and storage controls are needed to service the stored data and the lower is the cost per megabyte of disk storage needed to achieve a given level of system throughput. This strategy does, however, require a departure from traditional I/O subsystem guidelines, in particular, a departure from the objective

of always maintaining a 70 percent hit ratio in storage control cache.

Luckily, the target at which the strategy above takes aim appears to be broad and easy to hit. System-level performance is highly insensitive to the exact balance of the two key resources—storage control cache and processor buffer memory—needed to reduce disk delays. It is important only that some balance exists, with the larger part of the memory budget allocated in the processor and a small-to-moderate amount of memory provided in storage control cache.

Acknowledgment

The author is pleased to acknowledge the vital contributions of Charlie Werner and Neena Cherian toward the development of the tools that comprise SHARDS. Without such tools, the gathering of data such as that presented in this paper would not be possible.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references and note

1. B. McNutt and J. W. Murray, "A Multiple-Workload Approach to Cache Planning," *CMG'87, Computer Measurement Group Conference Proceedings, Internal Conference on Measurement and Performance Evaluation of Computer Systems,* Orlando, Florida (December 7–11, 1987), pp. 9–11.
2. B. McNutt, "An Overview and Comparison of VM DASD Workloads at Eleven Installations, with a Study of Storage Control Cache, Expanded Storage and Their Interaction," *CMG'89, Proceedings, International Conference on Management and Performance Evaluation of Computer Systems,* Reno, Nevada (December 11–15, 1989), pp. 306–318.
3. B. McNutt, "High-Speed Buffering of DASD Data: A Comparison of Storage Control Cache, Expanded Storage, and Hybrid Configurations," *CMG'90, International Conference for the Management and Performance Evaluation of Computer Systems,* Orlando, Florida (December 10–14, 1990), pp. 75–89. See particularly appendix result (A-9), which becomes Equation 6 of the present paper, when expressed in terms of miss ratios.
4. T. W. Ryan, "Optimizing Cache Performance," *CMG'90, International Conference for the Management and Performance Evaluation of Computer Systems,* Orlando, Florida (December 10–14, 1990), pp. 26–37.
5. G. Houtekamer, "Cache Management: Subsystem and Data Level Approaches," *CMG'91 Proceedings, International Conference for the Management and Performance Evaluation of Computer Systems,* Nashville, Tennessee (December 9–13, 1991), pp. 122–132.
6. Terminology is further confused by a conflict between this use of the term "holding time" and that used in References 2 and 3, in which "average holding time" is adopted as a nickname for "average residency time" and both phrases are used interchangeably. In this paper I avoid the term "holding time."
7. B. McNutt, "A Simple Statistical Model of Cache Reference Locality, and Its Application to Cache Planning, Measurement, and Control," *CMG'91 Proceedings, International Conference for the Management and Performance Evaluation of Computer Systems,* Nashville, Tennessee (December 9–13, 1991), pp. 203–210.
8. B. B. Mandelbrot, *The Fractal Geometry of Nature,* revised edition, W. H. Freeman & Co., New York (1983). See particularly p. 383.

Accepted for publication December 4, 1992.

Bruce McNutt *IBM ADSTAR, 5600 Cottle Road, San Jose, California 95193.* Mr. McNutt is an advisory engineer-scientist working in the area of DASD system performance evaluation. He joined IBM in 1983 and has worked in the area of DASD performance and workload characterization since that time. His 1990 paper "DASD Configuration Planning: Three Simple Checks" received the award for Best Management Paper at that year's conference of the Computer Measurement Group (CMG). Since first proposing the multiple workload approach to cache planning in 1987, he has continued to explore and develop the practical applications of this approach. Mr. McNutt received his B.S. degree in mathematics from Stanford University and his master's degree in electrical engineering and computer science from the University of California at Berkeley.

Reprint Order No. G321-5512.