

Tuning CICS/TS LSR Pools

“The Robin Hood Theory in Reverse”

March 9, 2006

By: Eugene S. Hudders

CICS/TS Performance Tuning Using C\TREK Course

This seminar covers a lot of tuning material and will require 8 hours a day. The total time for the seminar is 40 hours.

The followings topics included in this course are:

- Introduction to CICS Performance Tuning
- Using Operating System Information to Tune CICS/TS
- Tuning CICS/TS Processor Cycles
- Tuning Real Storage in CICS/TS
- Tuning Virtual Storage in CICS/TS
- Tuning CICS/TS Transaction Controls
- Tuning On Line VSAM Files
- Tuning NSR Files in CICS/TS
- Tuning CICS/TS LSR Buffer
- Tuning the Index CISZ
- Reviewing VSAM Free Space
- Reviewing the DB2 Interface

If you wish to attend this course, please send us an email at ctrek@actpr.com

This article covers several areas that should be considered when tuning LSR pools. Generally, the main performance objective when tuning CICS LSR pools is to improve the look-aside hit ratios. However, there are other tuning areas that are generally overlooked or not done. This presentation addresses the questions asked in many CICS tuning classes such as how many pools should I define, are there any advantages to defining multiple pools, how many strings should I define or how do I know that a file is monopolizing a pool and should be segregated into another pool? The article goes into detail not only on things that can be done to improve the look-aside hit ratios but also additional areas that can be tuned to improve the performance of the CICS LSR pools and improve response times. The performance-tuning product called C\TREK is used to display the information that must be gathered to tune the LSR pools when running under CICS/TS. Installations that have tuned their LSR pools using the techniques described in this article have been able to improve on-line response time and reduced the overall CPU being used by CICS.

There are two basic buffering techniques used by CICS, Non-Shared Resources (NSR) and Local Shared Resources (LSR). NSR buffering is covered in another article. In the case of LSR, data sets share resources, that is, they share common buffers and control blocks. This has a tendency of reducing the overall amount of resources needed to support files. In addition, look-aside occurs at all levels and is not limited to the index set records as in NSR. It is possible with proper buffering to avoid any physical I/O operations to satisfy a request and obtain a 100% look-aside hit ratio. In the past years, major VSAM enhancements within CICS require that the data set be assigned to LSR. However, there are some files that will work when defined as NSR but not when defined as LSR. For example, a non-Resource Level Sharing (RLS) LSR file that is used to browse for records (sequential read operations) is not allowed to issue a read for update in the middle of the sequential read operations. If a read for update is issued without ending the browse, the task will lock on the read for update request. The problem lies in the “shared” nature of LSR. The browse operation allows for the CIs to be shared, however, the read for update requires

exclusive control. As the task issuing the browse has control of the CI in shared mode, when the task asks for exclusive control of the CI, it is placed into an exclusive control wait waiting for the task that has the shared CI releases the CI. As the task has been put into a wait, it cannot execute to release the CI. These types of files must remain in NSR until the program logic is corrected.

Due to the better look-aside algorithm available in LSR, it is highly recommended that the vast majority of the installation's VSAM files be assigned to LSR buffering in CICS. This article reviews the areas that should be observed when tuning LSR files but not general VSAM tuning or programming techniques that would also be quite helpful in improving file performance in an on-line environment. The article focuses on generally known LSR buffer tuning areas such as initialization and buffer look-aside hit ratios and will also cover less known or sometimes overlooked areas when tuning VSAM files.

A file is assigned to LSR by specifying a value of 1 to 8 to the parameter "LSRPOOLID" when defining the VSAM data set under CEDD in CICS. The number indicates the resource share pool to which the file will belong. Entering "NONE" in this parameter places the file into NSR buffering. We will discuss the use of multiple pools later on in this article. For the time being, let us assume that we are dealing with files that have been assigned to LSR. As the values can range from 1 to 8, then there is a possibility of defining up to eight separate LSR pools within one CICS region. The resources assigned to one LSR pool cannot be shared with another pool. However, all of the files assigned to one pool share the resources defined to that pool. VSAM uses a simple Least Recently Used (LRU) algorithm when assigning resources. That is, whenever a buffer is needed, VSAM assigns the oldest unreferenced buffer to the file. The contents of the buffer are overlaid. We will not review Hiperspace buffers because Expanded Storage is no longer supported in the z/OS operating system structure. So, suffice to say that the content of the oldest unused buffer is lost. If the overlaid CI is needed again, then a new buffer must be assigned and the CI physically read from the data set. So, the trick to good look-aside hit ratios is to reference the buffer often to avoid being flushed out of the pool.

The LSR pools are built when the first file within a particular pool is referenced. If a file within a pool is opened during CICS start up, the LSR pool is built at that time. However, if no file is opened during CICS start up, the LSR pool is built when the first file in the pool is opened. Creating the pool when the file is opened delays the response time for that particular transaction. The resources assigned to the LSR pool can be determined dynamically or statically. In other words, you can let CICS determine the amount of resources to be assigned to the pool (dynamic allocation) or you can pre-specify the amount of resources to be assigned to the pool (static allocation). So, let us review the best alternative for LSR pool definition.

Dynamic LSR pool allocation is the easiest method of defining an LSR pool. When the first file assigned to the pool is opened, CICS will query all files assigned to the pool and allocate the necessary resources. The amount of resources allocated depend on an installation variable

called the SHARELIMIT to determine how much resources each file is going to contribute to the pool. The SHARELIMIT is a percentage and defaults to 50% but you can increase/decrease the amount in the LSRPOOL CEDA definition. We will use 50% for this discussion. CICS uses the number of strings assigned to the data set to determine the resource contribution. It is important to remember that there are some defaults that are observed for the data (BUFND) and index (BUFNI) buffers. The default data buffers (BUFND) are equal to the number of strings assigned to the file plus one. So, a file that has ten strings would require eleven data buffers. The default value for index (BUFNI) buffers is the number of strings. In the current case, the number of index buffers would be ten as there are ten strings assigned to the file. Another advantage of having the dynamic capability to create LSR pool is that it provides a fallback position (safety valve) in the case where files are added to the system with a different LSR pool id than the ones defined. In this case, CICS will allow the files to open and access without having to re-cycle the system.

Before continuing, we should identify what resources are needed to define an LSR pool. The first shared resource is the buffer space. CICS supports only eleven different CISZ sizes available for a VSAM file. The sizes are (in KB):

0.5, 1.0, 2.0, 4.0, 8.0, 12.0, 16.0, 20.0, 24.0, 28.0 and 32.0

Thus, if a file has a CISZ of 2.5 KB, then this data set would have to use a 4.0 KB buffer to accommodate this CISZ. Whenever the data set CISZ is not one of the eleven selected sizes, buffer fragmentation occurs. The fragmentation may or may not be acceptable. We will discuss buffer fragmentation later on in this article. CICS allocates buffers to one of the eleven possible sizes depending on the file's CISZ. Thus, it is possible to dynamically create a pool that does not have all eleven sizes available. Finally, an area that can be confusing is the trying to differentiate between the CISZ selected for a particular VSAM file versus the buffer size selected within CICS to process the CI. We use CISZ to describe the VSAM file "blocksize" definition and buffer size to describe the LSR pool area where the CI is to be processed.

A second shared resource is the number of strings allocated to the pool. Strings are allocated using the SHARELIMIT percentage, up to a maximum of 255 strings. The final shared resource is the key length. The key length specified has to be sufficiently large to accommodate the largest key belonging to the files in the shared LSR pool. A file with a longer key length than the one specified to the pool will result in the file not being able to be opened. Imagine that the longest key available is 100. If you try to open a file that has a key length of greater than 100 bytes, then file will not open.

Dynamic pool allocation is easy to implement and reduces the system programmer's intervention because there is no need to inventory the files to determine the largest key size, the

CISZ required or the number of strings needed. However, there are some major disadvantages to dynamically creating an LSR pool. The first disadvantage is that the dynamic allocation algorithm only creates one buffer pool that is shared between data and index buffers. Having only one buffer pool for both can create a contention between similar CI sizes in the data and index. In other words, mixing indices and data CIs in the same buffer pool has a tendency to flush indices out of the pool which tend to have a more concentrated access pattern because there are less index CIs than data CIs in files. This is especially true in a pool that contains heavily browsed files.

Another disadvantage is that the number of buffers selected is based on number of strings and not by file activity. You would need to increase the number of strings allocated in order to increase the number of buffers allocated to one particular CISZ. This technique would also increase the number of strings allocated up to a pool maximum of 255 wasting virtual storage that could have been allocated to other purposes such as additional buffer space. Finally, creating the pool dynamically takes a lot of time because every file in the pool must be queried to determine the key length and CI sizes of the data and the index. As the pool is created when the first file is opened, all the other data sets are closed and require CICS to perform an open to get the information from the catalog. A file open operation can take several seconds per file. The open process includes files that may or may not be used during the normal course of business. The response time for the task will be affected. You could define files in different LSR pools to be opened during initialization forcing the LSR pools to be built during start up to avoid the slow task response caused by opening the files on first use. The trade off is a slower CICS initialization. The real clunker is that the file is opened to determine the pool characteristics and then is closed. Thus, you will have to pay the price to open the file again when you reference it again. One of the first tasks in tuning the LSR pools is to determine how the pools were created ([Figure 1](#)).

```

FLGR ADVANCED COMPUTER TECHNOL  C\TREK ON-LINE          DATE 03/09/2006
APPLID CICSTS31                   APPLICATION DOMAIN    TIME 18:44:45
VERSION 6.4                       LSR POOL ANALYSIS   TERM 0207

TITLE  -POOL 1- -POOL 2- -POOL 3- -POOL 4- -POOL 5- -POOL 6- -POOL 7- -POOL 8-

ADDRESS 09B7F030 09B7F4B0 09B7F930 09BA6030 09BA64B0 09BA6930 09BA7030 09BA74B0
STATUS  ACTIVE   ACTIVE   INACTIVE INACTIVE INACTIVE INACTIVE INACTIVE INACTIVE
CREATED STATIC   STATIC
D/I BUF SEPARATE SEPARATE
STRINGS DEFINED  DEFINED
BUFFERS DEFINED  DEFINED
KEY DEF DEFINED  DEFINED
OPN ACB      7      3      0      0      0      0      0      0
SHARE %     100    100    50     50     50     50     50     50
KEY LEN     64     64     0      0      0      0      0      0
STR NO     64     64     0      0      0      0      0      0

CURSOR+ENTER LSR STATS CURSOR+PF4 LSR FILES PF5 MEMORY
4B :00.1 01/54
    
```

Figure 1. LSR Pool Start-Up Analysis

The best alternative is to pre-define the LSR pools through CEDA. This is called a static definition of the LSR pools. To obtain a static LSR pool definition you have to provide the buffer definitions, string and key length information. Buffer definition includes the number of buffers desired by size. If any of the elements are missing, then a dynamic allocation for the missing piece occurs. Static definition allows for a faster initialization because the files do not have to be queried. This avoids the overhead associated with having to open the files. Static definition results in a faster LSR pool initialization when the first file in the pool is accessed. In addition, you can now tune the buffer pool individually based on activity basis and you can also separate the data and the index buffers to avoid contention for the same CI sizes. You can also optimize the number of strings assigned to the pool. This is the recommended way to work with LSR in CICS.

Static definition requires system-programming intervention to determine the buffer sizes and quantity to be allocated, the number of strings required and the maximum key length. The process requires planning and exposes a system-programmer to errors. For example, if you forget to allocate a particular buffer size that a file requires, the file will use the next larger buffer size, if one is available. However, if there is no larger size, then the file will not open. Therefore, we always recommend that all pools have at least three 32 KB buffers defined to ensure that files will open even though performance is not optimum. The process to tune LSR pools is a repetitive process, that is, changes are made, the pool is re-initiated and the results measured.

LSR pool effectiveness is measured by the look-aside hit ratio. Generally accepted ratios are:

Data – 80%
Index – 95%
Combined – 93%

These objectives can vary but the important thing is to have objectives that can be used to measure the effectiveness of the LSR pools and the changes made. The index objective is higher because there usually are more I/O operations to the index component than to the data component. Most KSDS files have two or more index levels. Therefore, you would require two or more reads to the index component and only one read to the data component to locate a record.

Improving the hit ratio is usually a function of adding additional buffers to the particular size in question. The number of additional buffers depends on the amount of virtual and real storage available. Adding buffers to improve the look-aside hit ratio should concentrate on those buffers that have the highest request activity. For example, imagine that the 4 and 20 KB buffers reflect hit ratios of around 77% in the data pool ([Figure 2](#)). Suppose the 4 KB buffer pool has 1.4M requests while the 20 KB buffer pool has 529,000 requests. Both buffers should be improved. However, if there is a shortage of virtual and/or real storage, then fixing the 4 KB buffer would probably have a better effect on the overall LSR pool hit ratio. If there are sufficient resources available, then fix both. [Figure 3](#) provides the overall hit ratio for the index and data of two separate LSR pools.

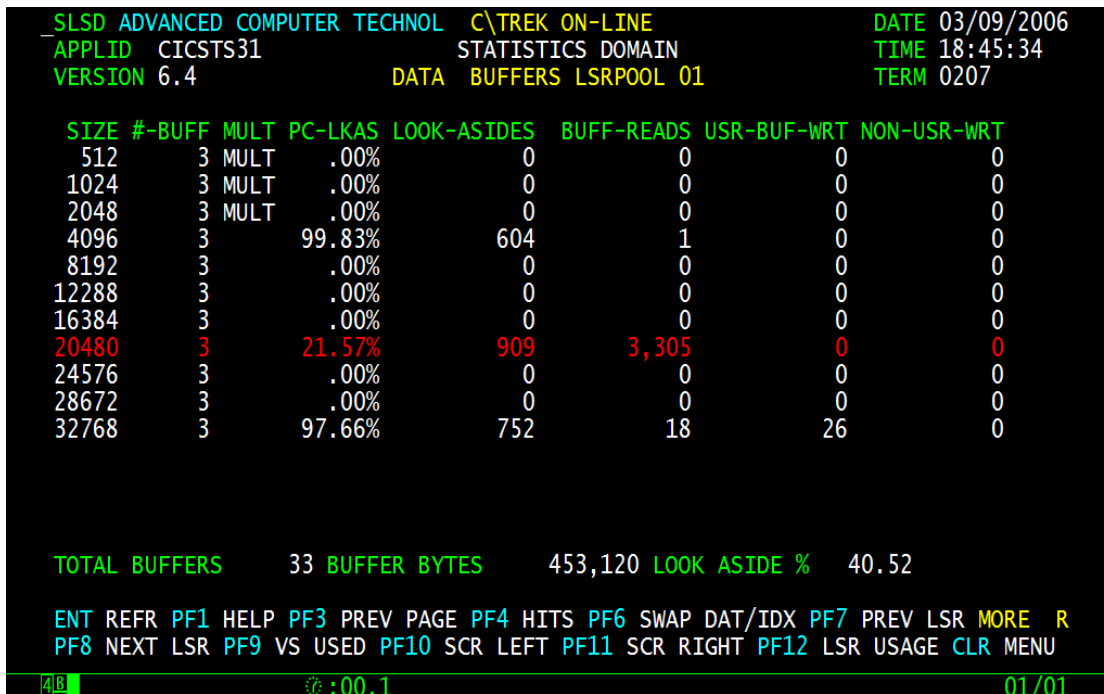


Figure 2. Buffer Look-Aside Hit Ratios

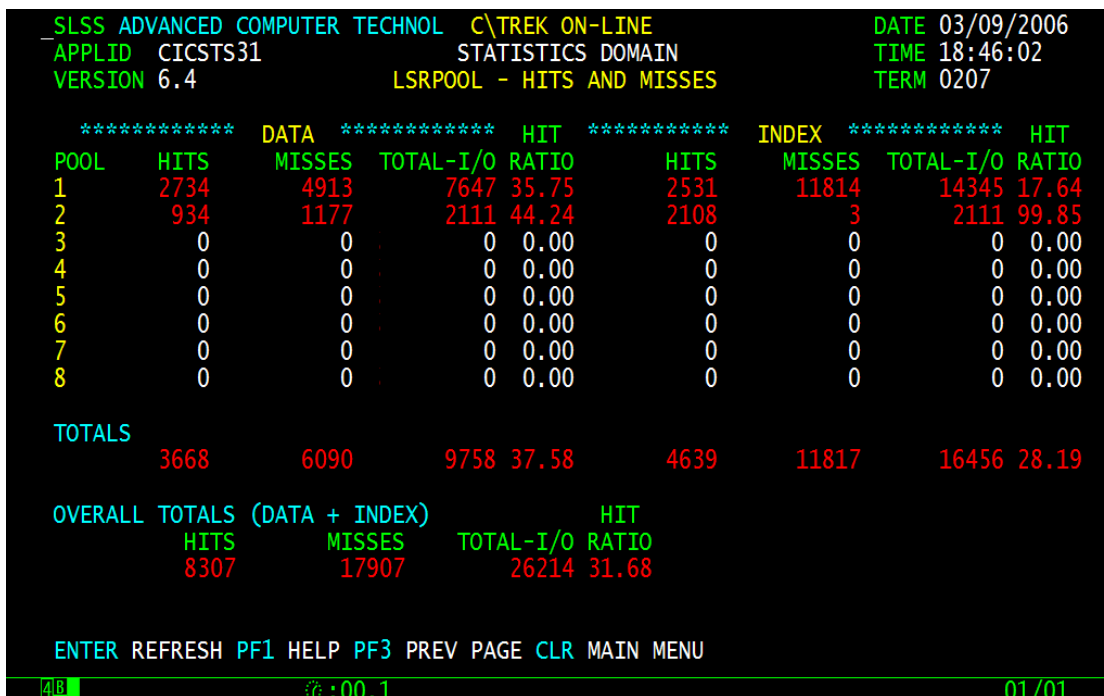


Figure 3. LSR Pool Summary Look-Aside Hit Ratio

Determining the number of buffers to add to the pool is usually done by trial and error. That is, you look at the different attainments and add a certain number to each of the buffer sizes that need adjustment. C\TREK provides a buffer pool recommendation that provides a starting point and assists in reducing the number of iterations required to determine how many buffers to add. The analysis is done by C\TREK includes a review of the actual attainments combined with the number of files in the pool. **Figure 4** provides a recommendation for the data buffers discussed. There are ten thousand 4 KB buffers and six hundred and fifty 20 KB buffers currently in LSR pool 1. The current allocation provided a look-aside hit ratio of around 77% for both buffer sizes, which is below the desired 80%. However, the recommendation provided in **Figure 4** to improve the look-aside hit ratio to 80% specifies that the 4 KB buffers should be raised to 12,500 while the 20 KB buffers should be raised to 813. A similar recommendation is also available for the index component.

```

FLSR ADVANCED COMPUTER TECHNOL  C\TREK ON-LINE      09B8D3D8  DATE 03/09/2006
APPLID CICSTS31                   APPLICATION DOMAIN  TIME 18:47:52
VERSION 6.4                       LOCAL SHARED RESOURCES  TERM 0207
                                OPTIMIZED
DATA  -POOL 1- -POOL 2- -POOL 3- -POOL 4- -POOL 5- -POOL 6- -POOL 7- -POOL 8-
CISIZ FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR
  512  0  0  3  8  0  0  0  0  0  0  0  0  0  0  0  0
 1024  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 2048  1  4  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 4096  1  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 8192  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
12288  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
16384  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
20480  5  6  0  0  0  0  0  0  0  0  0  0  0  0  0  0
24576  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
28672  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
32768  1  3  0  3  0  0  0  0  0  0  0  0  0  0  0  0
KL\ST 40 23 7 3 0 0 0 0 0 0 0 0 0 0 0 0
*****
FILES      8      3      0      0      0      0      0      0
NOSTR     23      3      0      0      0      0      0      0
BUFRS     16     11      0      0      0      0      0      0
BYTES    241664  102400  0      0      0      0      0      0
CURSOR+ENTER LSR STATS CURSOR+PF4 LSR FILES PF6 INDX/DAT PF12 MORE FUNCTIONS
  
```

Figure 4. Data Buffer Recommendation

LSR look-aside buffer hit ratios and percentages can be misleading. The look-aside percentage attained is for the buffer within the pool and not necessarily for the files that access the buffer. For example, a 4K buffer may reflect a look-aside hit ratio of 85%. This particular buffer size may be used by many files within the pool. So, some of the files can have a better look-aside hit ratio percentage than the buffer attainment while other files may have a lower look-aside hit ratio percentage. The file's look-aside hit ratio is based on the activity; the higher the activity, the better opportunity to get a better ratio. C\TREK can be used to determine the individual file look-aside hit ratios for both LSR and NSR.

Tuning the index buffer pools should take precedence over tuning the data pools because the index pool usually has more I/O activity. In addition, index CI sizes tend to be smaller than the data CI sizes. So, it is possible that a smaller investment in virtual storage would be required to improve the hit ratio. Also, there are less index CIs than there are data CIs, so the reference patterns for index records would be more concentrated.

Tuning data buffers can vary by system because the search patterns for the data component is generally random and access is disperse because of the size of the data component. Obtaining high hit ratios in the data component usually entails a large investment in virtual and real storage. Good data hit ratio candidates are files with a compressed access pattern that reference the data often, files with sequential read activity (browse) and files with read for update/rewrite and delete activity. Very large files with random activity can have a negative effect on the data pool reference pattern. VSAM files that have Share Options 4 specified, should not never be in the LSR pool because direct reads of the CI cause a re-read from the disk to ensure that we have the latest copy. This negates the look-aside capability of the LSR pool and tends to flush good buffers.

Once changes have been made, measure the results immediately after adding buffers to a pool. There has to be a certain amount of return on investment (ROI) for the virtual and real storage investment being made. The initial tuning ROI (1st time or after a major application installation) should be around 5% when the look-aside hit ratios are below 60%. If there is little ROI, then consider re-assigning these resources to another pool because availability of resources has a limit. Finally, there is a possibility of improving hit ratios by standardizing data CI sizes. Standard CI sizes will allow the creation of large buffer pools of a limited number of CI sizes. This allows for better resource utilization.

Tasks required for performance tuning of the LSR pools are:

- Maintain an inventory of files in each pool, identifying the associated CI sizes
- Reconciling the LSR buffers with the file CI sizes to ensure that the buffers are properly allocated
- Review the CICS statistics and ensure that the installation look-aside hit ratio percentages are achieved
- Ensure that there are sufficient strings available in the LSR pool
- Ensure that the LSR pools are statically defined via CEDA

There are other areas associated with tuning the LSR pools that are over looked in many installations. Many of these areas are overlooked because they are not visible or identifiable, there is insufficient manpower to dedicate to the tuning process or there is a lack of understanding as to their importance. In some cases the overlooked area is discovered as a

result of new applications and/or changes that occur. Some of the overlooked areas are as follows:

- Buffer fragmentation
- LSR buffer size versus file CISZ reconciliation
- Page boundary allocation
- Number of LSR pools defined
- Buffer pool monopolization
- Number of strings required
- Maximum Key Size

The amount of return on investment that can be received from tuning these areas will vary by installation. Some of these changes are minor but the idea is to have a well-tuned system. So, if you are going to look at the LSR pools with the intent of making them optimum, then fix everything that can be fixed. The first overlooked area that we will review is called buffer fragmentation. In general terms, buffer fragmentation is very common when using LSR pools in CICS. Buffer fragmentation occurs when the file CISZ is smaller than the CICS LSR pool buffer being used for the I/O operation. It is common because CICS uses only eleven buffer sizes to handle the I/O requests from VSAM files. VSAM has twenty-eight different CI sizes available. So, buffer fragmentation is inevitable unless you limit the VSAM cluster definitions to the sizes supported by CICS LSR. Buffer fragmentation results in extra virtual and real storage being used for I/O operations than the actual CISZ.

There are two basic types of buffer fragmentation: 1) Buffer fragmentation that results from selecting a CISZ that is not one of the eleven CICS buffer sizes such as selecting a CISZ of 1.5 KB but having to use a 2.0 KB buffer and 2) Buffer fragmentation that results in not having defined a particular buffer size that results in a CISZ using a larger buffer than required such as in the previous example having a CISZ of 1.5 K but using a 4.0 KB because we did not define a 2.0 KB buffer. The second type is definitely an area that requires attention. An interesting CISZ for a non-VSAM/E (non-Extended VSAM) file is 18 KB. In general terms, this CISZ provides the best track utilization (three 18 KB CIs per track or 810 KB per cylinder) and most records per cylinder. However, this CISZ causes buffer fragmentation because the appropriate CICS LSR buffer is 20.0 KB or a fragmentation of 2.0 KB. Therefore, there is a choice to be made of disk space versus buffer fragmentation. In this case, the buffer fragmentation is justified because of the positive disk space effect that results from using this CISZ. It should be noted that the 18.0 KB size is not as good a selection for VSAM/E (Extended VSAM) files because the space allocations are not as good. This caused by the need to add a 32-byte trailer to the physical record size causing that VSAM use three 6.0 KB physical records to handle the 18.0 KB CISZ. This results in a lower track utilization or 720 KB per track. This represents an 11% reduction in disk space use. This an important point to remember when converting from VSAM to VSAM/E.

One of the major problems associated with locating any buffer fragmentation is the number of VSAM files that may be assigned to a particular pool. An option used in some installations to

eliminate buffer fragmentation is to simply assign buffers to all eleven options. The problem with this alternative is that you may wind up allocating virtual storage to unused or low activity buffers, storage that could have been used to improve other buffers in the pool. So, the question would be how many buffers do you allocate to unused or low activity buffers without wasting too much storage? We have observed installations that are willing to allocate anywhere from three to twenty five buffers to address this condition. Let us analyze a hypothetical situation. Suppose that you have no files that use a 16 KB buffer and that you have five hundred 20.0 KB buffers defined. Assigning twenty 16.0 KB buffers results in an allocation of 320 KB of unused storage every day, every week and every month. If you never have a 16.0 CISZ, this becomes wasted storage.

However, would not it be better to take that 320 KB wasted storage and convert it to sixteen 20.0 KB buffers increasing the total to five hundred and sixteen buffers. The initial advantage is that the storage is used benefiting the 20 KB buffers immediately. If a 16.0 KB file is opened, you have part of the allocation in the 20.0 KB pool. That it was not the original allocation, so what? It was an estimate anyway. The important thing is to be able to identify the occurrence and take action. C\TREK provides a buffer to file CISZ reconciliation that quickly identifies buffer fragmentation and unused but allocated buffers (data and index) in the LSR pool. There is a possible exception to this condition. As long as there is a larger buffer size defined in the pool, the file will be able to open and be processed, even if it is not optimum. However, you must have a 32.0 KB buffer defined to handle opening large files. In this case, you would always define at least three (minimum allocation) 32.0 KB buffers as a safety valve, if a 32.0 KB were not required. C\TREK provides a buffer reconciliation routine that identifies if there is any buffer fragmentation as a result of not having defined the buffer size, which is the worst type of fragmentation ([Figure 5](#)). Missing buffer definitions are highlighted in red while buffer definitions that have no active file assigned are highlighted in yellow. Buffers highlighted in red need to be defined while those in yellow may be consolidated into other buffers.

```

FLCR ADVANCED COMPUTER TECHNOL  C\TREK ON-LINE      09B8D3D8  DATE 03/09/2006
APPLID CICSTS31                   APPLICATION DOMAIN  TIME 18:48:44
VERSION 6.4                       LSR POOL RECONCILIATION  TERM 0207
                                DATA
-PPOOL 1- -POOL 2- -POOL 3- -POOL 4- -POOL 5- -POOL 6- -POOL 7- -POOL 8-
FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR FLE BUFR
0 .5K 3 .5K 0 .5K 0 .5K 0 .5K 0 .5K 0 .5K 0 .5K
0 1K 0 1K 0 1K 0 1K 0 1K 0 1K 0 1K 0 1K
1 2K 0 2K 0 2K 0 2K 0 2K 0 2K 0 2K 0 2K
1 4K 0 4K 0 4K 0 4K 0 4K 0 4K 0 4K 0 4K
0 8K 0 8K 0 8K 0 8K 0 8K 0 8K 0 8K 0 8K
0 12K 0 12K 0 12K 0 12K 0 12K 0 12K 0 12K 0 12K
0 16K 0 16K 0 16K 0 16K 0 16K 0 16K 0 16K 0 16K
5 20K 0 20K 0 20K 0 20K 0 20K 0 20K 0 20K 0 20K
0 24K 0 24K 0 24K 0 24K 0 24K 0 24K 0 24K 0 24K
0 28K 0 28K 0 28K 0 28K 0 28K 0 28K 0 28K 0 28K
1 32K 0 32K 0 32K 0 32K 0 32K 0 32K 0 32K 0 32K

NOTE:
YELLOW = BUFFER DEFINED IN LSR POOL, BUT NO FILE ACTIVE FOR THIS CISZ.
RED    = ACTIVE FILE FOR THIS CISZ, BUT NO LSR BUFFER DEFINED.

CURSOR+ENTER LSR STATS CURSOR+PF4 LSR FILES PF6 INDX/DAT
4B | :01.5 | 01/54
    
```

Figure 5. Buffer Pool Reconciliation

Another similar consideration involves the many VSAM CI size options and buffer allocations. It is somewhat typical to see a 1.5 KB CISZ allocated for an index CI that has a 4.0 KB data size on a 3390 geometry disk drive. The LSR buffer required to process this CISZ is 2.0 KB. Is there any benefit to increasing the index CISZ so that the entire buffer is used? We can probably say yes to this question for most occasions although there may be some conditions that it may not be justified. The major problem with small CI sizes is that they tend to yield lower track utilization. So, for the data component, if you have a CISZ of 2.5 KB that requires a 4.0 KB LSR buffer to process, you may get better track utilization by raising the data CISZ to 4.0 KB and getting better track utilization. There factors to consider in this decision such as the record size and/or the need to have only one record per CI to avoid exclusive control conflicts. In the case of the index component, you may also benefit from raising the CISZ from 1.5 KB to 2.0 KB. The benefit may not be seen at the Sequence Set Level (lowest index level for a KSDS file) but at the Index Set Level (the second and higher levels in a KSDS file). There are several benefits that can occur from increasing the index CISZ to match the buffer size and eliminate buffer fragmentation such as:

- The larger index record may result in less index level records in the file. The less index set records, the less buffers are required to hold the entire high level index in virtual storage
- In some cases, a larger index set record may reduce the number of index levels a file has. The less index levels, the less index reads required to locate the data

- In some cases, having a larger index CI can reduce or eliminate potential key compression problems that result in loss of disk space and premature CA splits

This alternative requires the setting and control of CI sizes in VSAM cluster definitions. However, the benefits could include less index set records and less index level processing.

A very small benefit available in the LSR pool definition is the number of buffers allocated for the .5, 1.0 and 2.0 KB buffers. Buffers are allocated in 4.0 KB increments on a 4.0 KB boundary. Therefore, you could have inadvertently requested an incorrect multiple of buffers for these sizes resulting in fragmented buffer storage. When requesting storage for these particular sizes, make sure that the total number of buffers is a multiple of:

- 0.5 KB – multiple of 8
- 1.0 KB – multiple of 4
- 2.0 KB – multiple of 2

For example, imagine that you requested twenty-five 1.0 KB buffers. This would require 25 KB to accommodate the buffers. However, the buffer is allocated in page increments (4.0 KB). The actual allocation would be 28 KB or space for an additional 3 buffers without increasing the total allocated storage. So, when allocating storage for this range of buffers, always allocated in the appropriate increments previously stated (**Figure 6**).

```

_SLSD ADVANCED COMPUTER TECHNOL  C\TREK ON-LINE          DATE 03/09/2006
_APPLID CICSTS31                   STATISTICS DOMAIN       TIME 18:49:18
_VERSION 6.4                        DATA BUFFERS LSRPOOL 02  TERM 0207

  SIZE #-BUFF MULT PC-LKAS LOOK-ASIDES  BUFF-READS  USR-BUF-WRT  NON-USR-WRT
   512   3 MULT  43.51%    2,397      3,112         0           0
  1024   3 MULT   .00%         0         0         0           0
  2048   3 MULT   .00%         0         0         0           0
  4096   3     .00%         0         0         0           0
  8192   3     .00%         0         0         0           0
 12288   3     .00%         0         0         0           0
 16384   3     .00%         0         0         0           0
 20480   3     .00%         0         0         0           0
 24576   3     .00%         0         0         0           0
 28672   3     .00%         0         0         0           0
 32768   3     .00%         0         0         0           0

TOTAL BUFFERS      33 BUFFER BYTES      453,120 LOOK ASIDE %   43.51

ENT REFR PF1 HELP PF3 PREV PAGE PF4 HITS PF6 SWAP DAT/IDX PF7 PREV LSR MORE R
PF8 NEXT LSR PF9 VS USED PF10 SCR LEFT PF11 SCR RIGHT PF12 LSR USAGE CLR MENU
  
```

Figure 6. Unaligned Buffer Allocations

CICS has a capacity of eight separate LSR pools to be defined. These pools are known by a sequential number of 1 to 8. When LSR support was first added to CICS the search for a particular CI within the buffers was sequential. Thus, as you added buffers, the search for the record got longer and used more CPU. There was a point where the average search of these buffers exceeded the savings of finding the CI in the buffer pool instead of having to do an I/O operation. So, the use of different pools was justified to distribute the search cost across several pools reducing the average search cost. However, IBM improved the LSR search algorithm in the early 1990s and changed the algorithm from a sequential search to a hashing technique. Hashing provides a relatively even search cost regardless of the pool size. There is still a little additional overhead possible when having to handle synonyms, that is, CIs that hash to the same place. This is usually a minor cost. However, we now can allocate very large buffer pools without the search cost. In fact, IBM raised the number of possible buffers within a pool to 32K. So, the question is, do we need multiple pools and what are the benefits?

As a result of the new search algorithm and the capacity to allocate up to 32K buffers by buffer size, the need to allocate more than one pool has been reduced. Consolidating multiple buffers into one buffer pool can have some advantages because a larger number of buffers are available among the participants in the pool. The pattern varies by application. Access patterns to disk files are not simultaneous, that is, some files may be heavily accessed while others may be lightly used. This concept can be seen in the larger capacity disk drives available today. There was a fear many years ago that as you increased the disk storage capacity under one access mechanism that there would be contention. Disk drive capacity has increased dramatically and although contention is a concern, it is not based on having more files on a volume. Mixing application files into one pool allows VSAM to manage the resources based on activity. Thus, files that are lightly used “share” their buffers with more heavily used files.

Also, there is a tendency to add “a few more buffers” when defining a pool sort of like a safety valve. For example, you may determine that in a particular pool you may need four hundred 4.0 KB buffers. However, you may allocate 425 or 450 buffers to allow for growth or give you a small fudge factor. If you do this across the different buffer pools, you have over allocated the pools. Consolidating these pools gathers all the buffers and fudge factors into one large pool and lets VSAM allocate the buffers to those files that need it. There may be times when you may want to place a file in its own separate pool to ensure that the entire file is in virtual storage. This is the equivalent of a data table. This type of concept is used when you have a data table candidate but due to the amount of output I/O, the data table may not be a good performer. The quoted I/O figure is that a data table should be used for a file that has 90% or more read operations. So, if you have a file that has 70% read versus write operations, and then the data may not perform as well in a data table as leaving the file in LSR. This type of file is usually highly accessed and small enough to be a data table candidate. However, due to the high write to read ratio, it may not be a good candidate for a data table. In this case, assigning this particular file to its own LSR pool may be justified.

Another reason for having additional LSR pools has been to segregate files that monopolize an LSR pool **buffer**. The emphasis on the word **buffer** is intentional because a file monopolizes a buffer within a pool and not the pool as commonly seen in other documentation. However, what CICS statistic do you use to determine that a file is monopolizing a particular LSR buffer? File statistics provided by CICS are regarding the number of I/O requests not the number of buffers used. Intuitively one could reach the conclusion that I/O requests equal number of buffers and it could be wrong. Take for example a single key-point record file that may consist of 100 records. This file could reflect millions of requests a day but would only use 100 buffers maximum. Another example would be a heavily browsed file. In this case, the number of browse requests does not take into consideration the number of look-aside hits that you had in the same CI before you had to use another buffer for the next CI. C\TREK provides a file buffer allocation display that indicates how many buffers a file in the LSR pool is using (Figure 7).

```

FLSB ADVANCED COMPUTER TECHNOL  C\TREK ON-LINE      20B16B60  DATE 03/09/2006
APPLID CICSTS31                   STATISTICS DOMAIN  TIME 18:50:42
VERSION 6.4                       FILES IN LSR POOL 02  TERM 0207

POOL BUFFER TYPE DATA          ----POOL INFORMATION----  BUFFER SIZE          512
OPEN ACB                          3  STRINGS IN POOL          64  MAX POOL KLEN        64
VSAM VSRT          08CC74E0  DATA BUFF HDR    20B16B60  INDEX BUFF HDR    20B87B60
                                -POOL BUFFER INFORMATION-
BUFFER      ADDR 20B16B60  NUM OF BUFFERS          3  STRINGS USED          0
BUFFERS     USED  0  STRGE ASSIGNED         1536  HITS                   2626
BUFF START ADDR 20B05A00  READS                   3380  TOTAL BUFF USE        3
USER WRITES  0  NON-USER WRITES          0  % BUFFERS USED        100.0
DDNAME  BUFUSE  AMB-ADDR  BUF-ADDR  DSNAME
VSAMFLEB  2  20BFC028  20B05A00  PRGMLX1.VSAMFLEB.CICS.INTERNAL.DATA
VSAMFLE1  1  20BF99E8  20B05C00  PRGMLX1.VSAMFLE1.TEST.INTERNAL.DATA

PF6 SWAP D/I PF7 BACK PF8 FORW PF9 LEFT/RIGHT PF12 MORE OPTIONS  MORE. L

```

Figure 7. File Buffer Use Within Pool

Many performance monitors do not reflect the number of buffers actually used by a file and this is the information needed to determine which if any file is monopolizing a buffer in a pool. C\TREK provides buffer use information so that you could relate the number of buffers used to total I/O requests by file. However, there is one other thing that must be done. You also have to define the word “monopolizing”. What do you consider a monopolizing percent? 90? 70%? 50%? 20%?

We need to review certain concepts before we can address LSR buffer monopolization. First of all, all files in a CICS system are not accessed at the same time. In fact, studies have shown that less than 20% of the defined files within a CICS region are continuously used. This is similar to the “old” inventory 80/20 rule where 20% of the inventory represents 80% of the activity. In other words, the 20% may represent your installation’s most important files. Therefore, the 20% can be called your “bread and butter” files. Second, what is LSR? It is a means of being able to **share** resources. The algorithm used by CICS to allocate buffers is a Least Recently Used (LRU) formula. So, the buffers that contain data are holding the most recently reference CIs. Your “bread and butter” files probably populate the majority of these buffers. Don’t you need fast access to provide good response times for these files? So, what is wrong with having them “monopolize” the buffer pools?

LSR is really a Robin Hood Story in reverse. In LSR you use the resources of lightly or intermediately used files by the higher activity files. In other words, “**you rob from the poor to give to the rich**”. You take resources from low activity files to give to high activity files. If your response times are good, then why should you worry because you had to do an I/O for one of the 80% of your files that has less activity? Their participation in LSR is to contribute resources. It sounds sort of cruel but that is the truth. Think of it another way. If you had placed this low activity file into NSR to “protect” its resources, you would have virtual storage allocated to the file at the expense of being able to provide this storage to your “bread and butter” files.

So, when is buffer monopolization a problem? The best response is, when you are not receiving good response times from your important files while achieving a high look-aside hit ratio. This condition usually manifests itself in one or two files controlling more than 80% of the buffers affecting other high activity or important files. A possible solution is to add more buffers even though you may be achieving the look-aside hit ratio. The idea is that eventually the files in the buffer pool will balance the buffer usage. The other possibility is to move one or both of these files to their own pool.

Another area that requires attention is the number of strings assigned to an LSR pool. The number of strings required depends on the pool activity and the type of operations being performed. The first consideration is the activity against the files. Some activity only requires one string but other activity that involves AIX files will require additional strings especially if it is an upgrade AIX. Certain I/O requests release the string immediately such as a direct read. However, browse or read for update operations will hold the string for a longer period of time. The number of strings required is also affected by how well the look-aside hit ratio is working because the better the look-aside hit ratio, the less physical I/O operations that require a string are performed. The number of strings assigned is generally over allocated. So, before increasing the number of strings in a pool, ensure that you are attaining the look-aside hit ratio for both the index and the data components. The higher the attainment, the less physical I/O requirements for an I/O operation. CICS provides information about how many strings are allocated, how many are currently in use, the peak number of strings, and the number of times

you had to wait for a string and how many are currently waiting for a string. You want to have zero wait on strings. The objective is to have the peak number of strings used to be around 40 to 60% of the total allocated. C\TREK provides a string analysis and recommendation in this area.

It is important to note that there are two types of string waits associated with files assigned to LSR. One is a string wait caused by having insufficient strings assigned to the entire pool to handle the concurrent requests that can occur at any given point. This condition was discussed in the previous paragraph. The second condition is associated with the number of strings that can be assigned to the file at any one moment. This type of short on strings condition would be reflected at the file level and not at the LSR pool level. The number of strings specified for a file represents the maximum number of concurrent requests allowed for that file at any given moment. As with all short on strings conditions, make sure that the proper look-aside hit ratios are being achieved for the associated buffer sizes. If not, adjust the buffers first to get the improved look-aside hit ratio. C\TREK provides a string analysis that can provide a recommendation as to the number of strings that should be defined for the pool (Figure 8).

```

_LSR ADVANCED COMPUTER TECHNOL  C\TREK ON-LINE          DATE 03/09/2006
APPLID CICSTS31                   APPLICATION DOMAIN      TIME 18:51:24
VERSION 6.4                       LSR STRINGS RECOMMENDATIONS  TERM 0207

```

POOL	NUM-OF	CONC-ACT	RECOMM.	
	STRINGS	STRINGS PERCENT	STRINGS	PERCENT
1	64	6 9.37	32	18.75
2	64	3 4.68	32	9.37
3	0	0 0.00		
4	0	0 0.00		
5	0	0 0.00		
6	0	0 0.00		
7	0	0 0.00		
8	0	0 0.00		

ENTER REFRESH PF1 HELP PF3 PREV PAGE CLR MAIN MENU

Figure 8. LSR Pool String Recommendation

Associated with the number of strings specified is the key length assigned to the pool. Fixing the specified pool key length is a minor point because not a lot of storage is involved. The key length has to be at least as large as the longest key of any file opened in the pool. Many installations simply specify a key length of 255 bytes. This specification results in a waste of virtual storage in most installations because it is rare to see a key greater than 128 bytes long.

The key length is used during each I/O operation in which a string is involved. Therefore, if both strings and key length are over allocated, the resulting storage reflects some waste. Reduce the key length to 128 or lower and use the saved for additional buffers.

In closing, let us review which files are candidates for LSR. It is probably a lot easier to identify which files are not candidates. For example, a Share Options 4 file should never be assigned to LSR because there is little look-aside that can occur on these files. Keeping a Share Options 4 file in an LSR pool tends to flush the buffers because the assigned buffer becomes the most recently used and will not be reused. There are some highly active third party package control files that are specified Share Options 4 and can have devastating effects on your LSR pool statistics. Another type of file is one that does not follow Command Level programming guidelines. An example is a file that in the middle of a browse operation (READNEXT) receives a read for update request. Although this function works while the file is assigned to NSR, in LSR (non-RLS) the task will hang and can cause problems that affect the entire system. In NSR, a second string is assigned to handle the read for update request and the CI is re-read into a new buffer assigned to the second string. This, the CI would appear twice in storage. Another file type that could cause problems for an LSR pool is one that has a lot of CA splits because the buffers are not reused. These types of files should be tuned separately to reduce the number of CA splits, if you want to keep them in LSR. In the near future, a separate article will be available on tuning VSAM files and will address CI/CA splits.

However, it should be noted that CA splits under LSR do not tie up the CICS TCBs while performing the CA split. LSR uses synchronous file requests and the UPAD exit to handle CI and CA splits. This method does not tie up either the subtask or main task TCBs. VSAM takes the UPAD exit while waiting for a physical I/O to complete, allowing for processing to continue and let CICS dispatch other work. NSR file control requests are done asynchronously and cause the CICS main task or subtask to wait during the split. It is highly recommended that the VSAM subtask be activated if you have NSR files prone to CI/CA splits. Effective use of the VSAM subtask (CO TCB) requires that CICS be running on a multiprocessor.

If you have any questions, comments or request for more information, please contact us:

Address:	C\TREK Corporation PO BOX 560069 MONTVERDE FL 34756	Phone:	(407) 469-3600 C\TREK Corporation (787) 756-5620 Advanced Computer Technology
E-mail:	ctrek@actpr.com	Fax:	(787) 756-5150
Website:	www.ctrekcorp.com	Support::	(787) 397-4150 (321) 297-5838 (787) 462-0406