SEGUS

White Paper

DB2 and Memory Exploitation

Fabio Massimo Ottaviani - EPV Technologies

1 Introduction

For many years, z/OS and DB2 system programmers have been fighting for memory: the former to defend the limited amount of memory available on the system, the latter to try to enlarge the DB2 pools in order to improve performance.

Thanks to the technological evolution, this fight should no longer make sense. A huge amount of memory is now available to the z/OS systems and subsystems and, with the z13 machines, the memory price has also become three times cheaper than previous machine generations.

However, old habits are very difficult to change, so it's still pretty common to see DB2 subsystems suffering for lack of storage—even if a lot of memory is available, and unused, in the z/OS system.

It's important to be aware that DB2 memory exploitation can provide:

- better application performance;
- reduced I/O activity;
- reduced CPU consumptions.

In this paper, we will discuss the available measurement which can be used to understand if DBD, Skeleton pools, Global Dynamic Statement Cache and Buffer Pools performance can be improved by exploiting the available memory.

2 DBD Pool

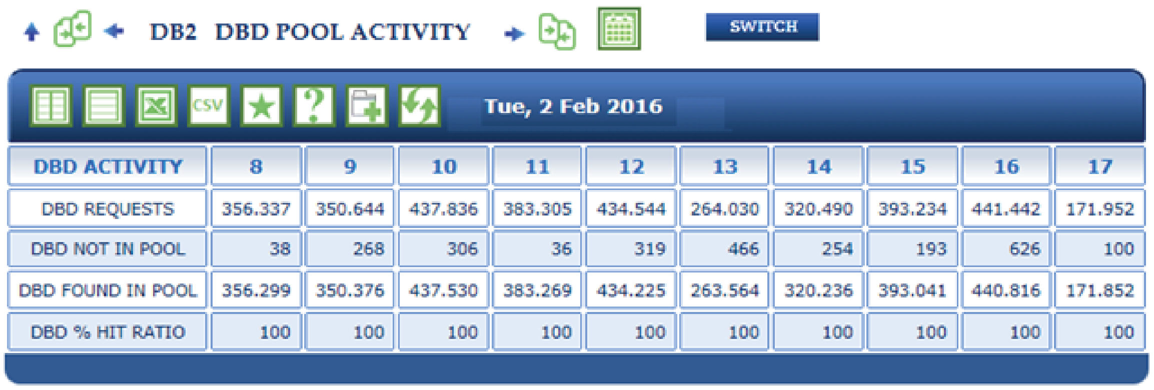
Database descriptors (DBDs) are the internal representation of DB2 database definitions.
A DBD normally includes many objects (e.g. table space, index); the number of objects determines the DBD size.

When an SQL statement is executed, the referenced DBDs need to be available in the DBD pool; the following situations may occur:

- DBDs are already in the DBD pool; it's a hit; this is the best option for application performance; no I/O operations are required and there is no application delay;
- DBDs are not in the DBD pool; it's a miss; DBDs have to be loaded into the pool, performing some I/O operations with consequent application delay;
- DBDs are not in the DBD pool; if the pool is full and the LRU algorithm is not able to free any space for the new DBDs because all the DBDs in the pool are active, a DBD failure occurs and the application ends in error.

DBD failures should be avoided; but if they do happen, it's strongly advised that you increase the DBD pool size. The number of failures is provided in the QISEDFAL metric.

For the hit ratio, you should aim for values higher than 95% (as close as possible to 100%). In the Figure 1 table you see an example of a well-sized DBD pool.

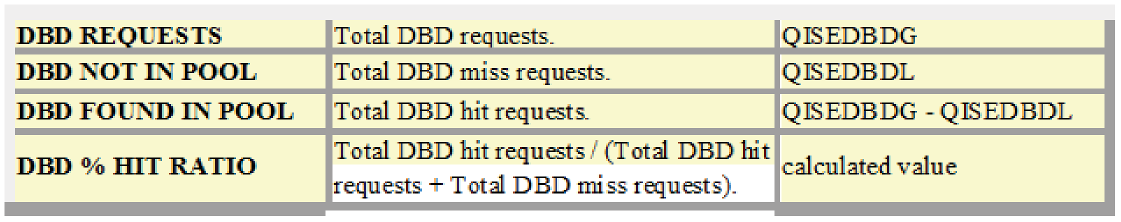


DBD ACTIVITY	8	9	10	11	12	13	14	15	16	17
DBD REQUESTS	356.337	350.644	437.836	383.305	434.544	264.030	320.490	393.234	441.442	171.952
DBD NOT IN POOL	38	268	306	36	319	466	254	193	626	100
DBD FOUND IN POOL	356.299	350.376	437.530	383.269	434.225	263.564	320.236	393.041	440.816	171.852
DBD % HIT RATIO	100	100	100	100	100	100	100	100	100	100

Figure 1

The DBD pool in this example has a size of more than 180,000 pages and it's used more than 95%.

All the metrics used in Figure 1 are provided by the DB2 statistics trace and collected in IFCID 2 (SMF 100). They are[1]:



DBD REQUESTS	Total DBD requests.	QISEDBDG
DBD NOT IN POOL	Total DBD miss requests.	QISEDBDL
DBD FOUND IN POOL	Total DBD hit requests.	QISEDBDG - QISEDBDL
DBD % HIT RATIO	Total DBD hit requests / (Total DBD hit requests + Total DBD miss requests).	calculated value

Figure 2

1 Table from the EPV for DB2 Help System

The number of 4K pages allocated for the DBD pool is provided by the QISEDPGE metric, while the number of free pages is provided by QISEDFRE.

The percentage of DBD pool utilization can be calculated as:

```
QISEDPGE - QISEDFRE
---------------------------------
QISEDPGE
```

High DBD pool utilization: not an issue, unless combined with failures or low hit ratio. Low DBD pool utilization: indicates a waste of storage.

3 Skeleton pool

The Skeleton pool caches Skeleton Cursor Tables (SKCT) for plans and Skeleton Package Tables (SKPT) for packages, in order to speed up SQL statement executions.
In this document we will only discuss SKPT because these days they are much more relevant for application performance[2].

When an application needs to execute a package, the required sections of the package have to be available in the Skeleton pool[3]; the following situations may occur:

- package sections are already in the Skeleton pool; so it's a hit; this is the best option for application performance; no I/O operations are required and there is no application delay;
- package sections are not in the Skeleton pool; so it's a miss; they have to be loaded into the pool performing some I/O operations with consequent application delay;
- package sections are not in the Skeleton pool; if the pool is full and the LRU algorithm is not able to free any space for the new sections because all the objects in the pool are active; a Skeleton pool failure occurs and the application ends in error.

Skeleton pool failures should be avoided; if they do happen, it's strongly advised that you increase the Skeleton pool size. The number of failures is provided in the QISEKFAL metric.

For the hit ratio you should aim for values higher than 95% (as close as possible to 100%). In the table in Figure 3 on the next page, you see an example of a badly-sized Skeleton pool.

Note that the Package Table hit ratio is too low. Values lower than 95% are highlighted with a pink background.

2 Since DB2 V10 all DBRMs have to be bound into packages.
3 A thread-specific copy of the package will then be created into the agent local pool storage; if a given package is allocated to 5 threads, there will be 5 copies of the package in the agent local pool storage of the DBM1 AS

EDM ACTIVITY	8	9	10	11	12
CT REQUESTS	806.139	732.853	787.458	677.092	609.382
CT NOT IN POOL	17.444	15.575	14.168	13.453	12.781
CT FOUND IN POOL	788.695	717.278	773.290	663.638	596.602
CT % HIT RATIO	98	98	98	98	98
PT REQUESTS	29.223.619	25.861.610	30.092.847	23.316.594	20.984.935
PT NOT IN POOL	3.747.422	3.122.852	2.928.491	2.926.664	3.336.396
PT FOUND IN POOL	25.476.197	22.738.758	27.164.357	20.389.931	17.648.538
PT % HIT RATIO	87	88	90	87	84

Figure 3

The Skeleton pool in this example has a size of only 4,000 pages and it's utilization is close to 100%. It's size should be increased in order to improve the hit ratio.

All the metrics used in Figure 3 are provided by the DB2 statistics trace and collected in IFCID 2 (SMF 100). They are[4]:

FIELD	DESCRIPTION	SOURCE
CT REQUESTS	Total Cursor Table requests.	QISECTG
CT NOT IN POOL	Total Cursor Table miss requests.	QISECTL
CT FOUND IN POOL	Total Cursor Table hit requests.	QISECTG - QISECTL
CT % HIT RATIO	Total Cursor Table hit requests / (Total Cursor Table hit requests + Total Cursor Table miss requests).	calculated value

FIELD	DESCRIPTION	SOURCE
PT REQUESTS	Total Package Table requests.	QISEKTG
PT NOT IN POOL	Total Package Table miss requests.	QISEKTL
PT FOUND IN POOL	Total Package Table hit requests.	QISEKTG - QISEKTL
PT % HIT RATIO	Total Package Table hit requests / (Total Package Table hit requests + Total Package Table miss requests).	calculated value

Figure 4

The number of 4K pages allocated for the Skeleton pool is provided by the QISEKPGE metric, while the number of free pages is provided by QISEKFRE. The percentage of utilization of the Skeleton pool can be calculated as:

$$\frac{\text{QISEKPGE - QISEKFRE}}{\text{QISEKPGE}}$$

High Skeleton pool utilization is not an issue unless combined with failures or low hit ratio. Low Skeleton pool utilization indicates a waste of storage.

4 Table created from the EPV for DB2 Help System

4 Global Dynamic Statement Cache

In recent years the use of dynamic SQL has grown dramatically in the DB2 world.

One of the most important characteristics of dynamic SQL is that for a dynamic SQL statement to run, DB2 must determine an access path and build the skeleton control structure (SKDS). This is done through the PREPARE statement.

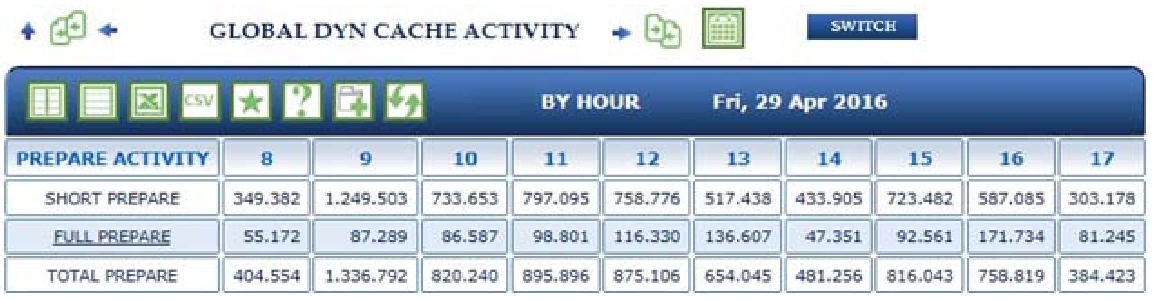
The PREPARE process is generally long and consumes CPU, so it can be an important issue when the amount of dynamic SQL is very high.

A dynamic statement has to be prepared the first time it runs—there's no way to avoid that—but the prepared statement can be re-used in all subsequent executions, if it can be saved somewhere. The Global Dynamic Statement Cache (GDSC) was introduced in DB2 V5 for this purpose.
The GDSC has to be enabled by setting ZPARM CACHEDYN to YES, while the size of the cache is specified with ZPARM EDMSTMTC.

The size of GDSC, limited for years because of 31bit virtual storage constraints, can be now very large but at many sites it is still set too small to allow a full exploitation of its potential.

When a statement has to be prepared, DB2 will search for the statement in the GDSC; the following situations may occur:

- the same statement is found; it will be copied to local thread storage; this is called Short Prepare; as you can imagine it requires a lot less time and effort;
- the same statement is not found; the complete prepare process has to be run; this is called Full Prepare.

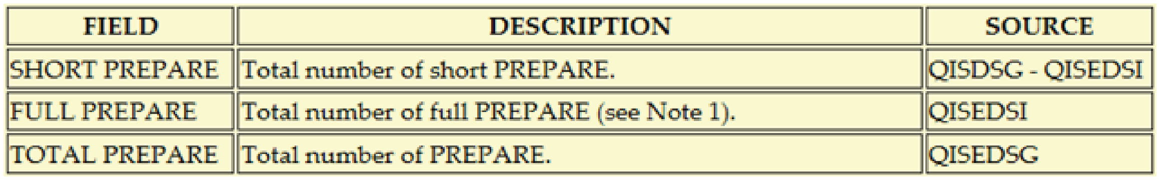


PREPARE ACTIVITY	8	9	10	11	12	13	14	15	16	17
SHORT PREPARE	349.382	1.249.503	733.653	797.095	758.776	517.438	433.905	723.482	587.085	303.178
FULL PREPARE	55.172	87.289	86.587	98.801	116.330	136.607	47.351	92.561	171.734	81.245
TOTAL PREPARE	404.554	1.336.792	820.240	895.896	875.106	654.045	481.256	816.043	758.819	384.423

Figure 5

In the example in Figure 5 on the previous page, you can appreciate the benefits provided by the GDSC. As you can see, a lot of Full Prepare processing has been avoided. The Short Prepare number is generally much higher than that for the Full Prepare.

All the metrics used in Figure 5 are provided by the DB2 statistics trace and collected in IFCID 2 (SMF 100). They are[5]:



FIELD	DESCRIPTION	SOURCE
SHORT PREPARE	Total number of short PREPARE.	QISDSG - QISEDSI
FULL PREPARE	Total number of full PREPARE (see Note 1).	QISEDSI
TOTAL PREPARE	Total number of PREPARE.	QISEDSG

Figure 6

The hit ratio can be estimated by dividing the number of Short Prepare by the total number of Prepare.

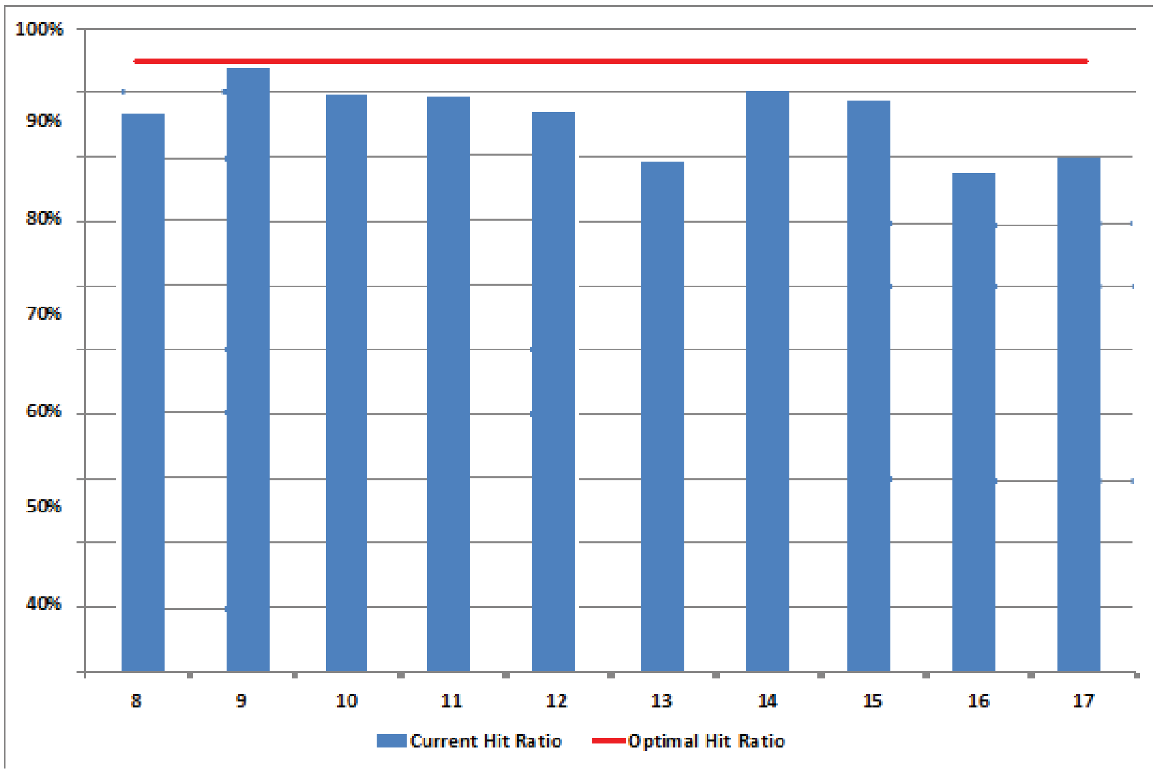


Figure 7

The GDSC hit ratio is reported in Figure 7. The red line, set at 95%, represents a commonly used best practice for the GDSC hit ratio minimum value. As you can see, it is never reached.

5 Table from the EPV for DB2 Help System

In this case, a possible reason is that the GDSC is too small. It has a size of only 40,000 pages and it is fully used, so the suggestion would be to double it (it will require only 160 MB of storage), and analyse the effects on the hit ratio.

However in some situations the main issue is the statement format, not the size of the GDSC.
For a statement in the GDSC to be re-used, it should be *exactly* the same, down to the smallest detail (including literal values, spaces, order, authorization id, bind options)[6].

Generally, the use of literals is the most likely reason why a statement in the GDSC can't be re-used. The best solution should be using parameter markers instead of literals, but it will require modifying the application.
A possible alternative is the CSWL (Concentrate Statement With Literals) PREPARE parameter, introduced with DB2 V10. When using CSWL, the statement is stored into the cache and each literal is replaced with &[7].

For example:

```
SELECT COL1, COL2, COL3 from Table WHERE COL1 = 'A'
```
is stored in cache as:
```
SELECT COL1, COL2, COL3 from Table WHERE COL1 = &
```
If the following subsequent statement has to be prepared:
```
SELECT COL1, COL2, COL3 from Table WHERE COL1 = 'B'
```
DB2 will find a statement in the dynamic statement cache containing the replaced character (&) and will re-use it.

CSWL doesn't require more storage. Conversely, it can reduce the storage wasted by statements which differ only in the literals.

5 Buffer Pools

Buffer pools are a key element in DB2 application performance.
Their goal is to allow the application to get data and index pages from storage, without requiring I/O operations.
Tuning buffer pools is a very complex topic and it will not be discussed completely here. In this paper, we will only talk about the possibility of improving application performance by increasing the buffer pool size and by exploiting buffer pool long term page fixing.

When an application issues a getpage request, the page is searched for in the buffer pool correspondent to the tablespace or indexspace where the page resides. If the page is not found (buffer pool miss), the application is suspended and a synchronous I/O is needed to load that page into the buffer pool.

6 See DB2 V11 for z/OS Managing Performance.

7 This process doesn't occur if the statement is coded with the "?" parameter marker.

In SMF 101 IFCID 3 the following metrics are provided:

- QWACAWTI; total wait time for synchronous read;
- QWACARNE; total wait events for synchronous read

QWACAWTI tells you what the impact of a buffer pool miss is on application performance.
By dividing QWACAWTI by QWACARNE you can evaluate the average response time for synchronous read operations in order to understand if there are I/O performance issues.

Another important characteristic of synchronous read operations is that the CPU consumed to execute the I/O is charged to the application and it is standard CPU time, not zIIP time as it happens for asynchronous read operations charged to the DBM1 address space.
A recent IBM study estimated 35 CPU microseconds (on a 2827-712) per DB2 synchronous read. Based on that, you can estimate about 40 MIPS used every 1,000 read per second.[8]

In the Figure 8 below, we show an example of very high read rate to a couple of buffer pools on a two member data sharing group.[9]

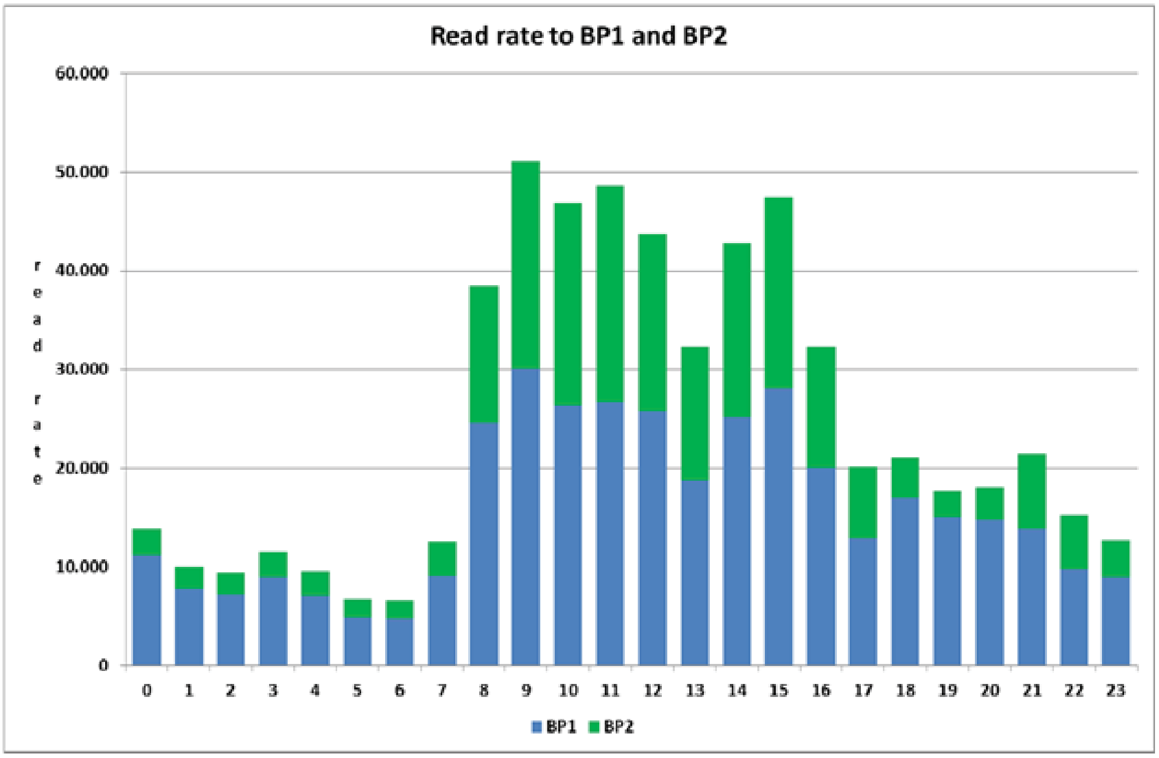


Figure 8

The size of BP1 and BP2 was about 200,000 pages each on both members.
After showing this graph to the DB2 system manager, he decided to double the size of both buffer pools in both members, "investing" about 3.2 GB of storage.

8 "System z: Advantages of Configuring More Memory for DB2 Buffer Pools" V2.0 - February 2015

9 The metric used in the graph is QBSTRIO provided in SMF 100 IFCID 2.

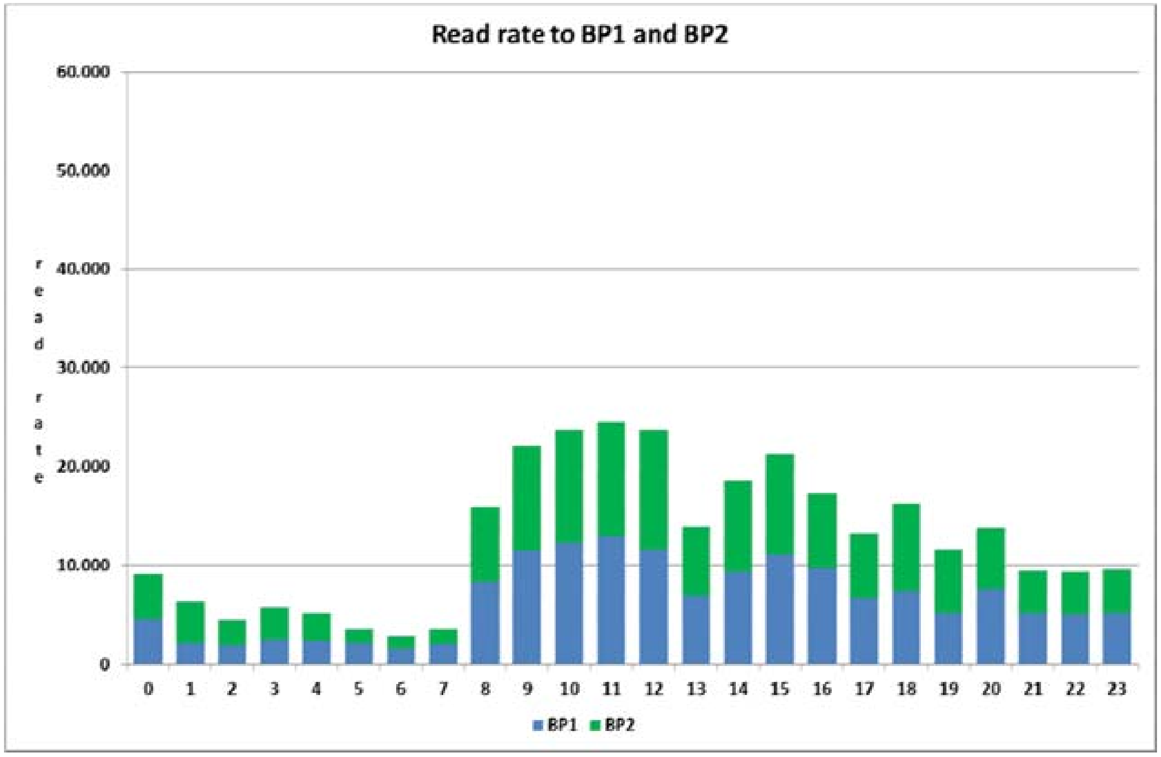


Figure 9

The result is reported in Figure 9.

At least 20,000 reads per second have been saved. About 50% were synchronous so the estimated saving is about 400 MIPS.

It's not guaranteed that by increasing the buffer pool size you will be able to get a significant reduction of read operations. This is sometimes due to specific workload characteristics.[10]

In this situation, you can try to minimize the impact read operations have on performance by exploiting the PGFIX buffer pool parameter. If you set PGFIX=YES, the buffer pool is fixed in real storage so the page fix/unfix operations are no longer required when performing a read to the buffer pools—with significant benefits in terms of CPU usage.

If 1MB pages or 2GB pages are available in the system LFAREA (Large Frame Area), DB2 will automatically use them for the buffer pools where PGFIX=YES is specified. This will provide additional performance benefits and CPU savings due to quicker virtual address translation operations.

10 In DB2 V11 you can use simulated buffer pools to measure how many read I/O operations can be avoided if the buffer pool size is increased.

6 Conclusions

In modern z/OS systems memory is normally abundant but often only partially used.

Many available DB2 functionalities allow for the exploitation of this available memory with big benefits for application performance and reduced CPU consumption.

SEGUS Inc is the North American distributor for EPV products

For more information regarding EPV for z/OS, please visit www.segus.com
or call (800) 327-9650