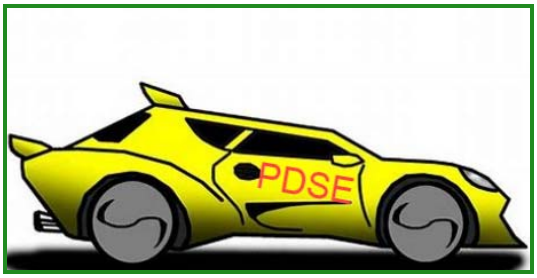
Optimizing PDS and PDSE Performance



A while back we were working with a client that was evaluating moving from full capacity to sub-capacity CPCs. As part of the preparation for the move, we helped them evaluate workloads that consume large amounts of CPU time and therefore could potentially be impacted by the slower speed CPs of the sub-capacity models. We also took the opportunity to make sure that the systems overall were running as efficiently as possible.

To that end, one of the things we looked at was the use of PDSEs. PDSEs had a few teething problems in their early days, and as a result, many customers shied away from using them. However, that was a *long* time ago, and PDSE has matured into being a responsible citizen of the z/OS ecosystem. In fact, the COBOL V5 and later or the PL/I V5 and later compilers *require* PDSEs for the program objects they create. Additionally, if you look at your system volumes, you might be surprised to see how many software product data sets are now in PDSE format. So, PDSEs are not going away. In fact, if you want any of the more recent enhancements such as partitioned data set encryption support or member generations, they are *only* available for PDSEs.

As a result of all this, we felt that this would be a good time for a refresher article about partitioned data set performance in general, and PDSEs in particular, focusing on how they can be used to improve the performance and efficiency of your systems. This article was further inspired by SHARE in New Orleans 2023 Session 31030, *PDSE Pitstop: Performance, Problem Analysis, and Pheatures*, by **Tabari Alexander** and **Trevor Geisler**, and we want to thank them for all their help and patience in the creation of this article. I also want to thank the indomitable **Peter Relson**, without whose help and patience this article would not be possible, and my colleague **Mario Bezzi** for his many suggestions that resulted in a much more valuable article.

Target Audience

This series of articles is aimed at performance analysts, system programmers, and anyone who is responsible for a product or application that has PDS or PDSE data sets that will be heavily used. Their objective is to focus on the things that you can do to optimize the performance of applications that use those data sets.

We initially expected this article to be maybe 10 pages long. But the more we researched, the more we discovered things that are not widely known, so 10 pages quickly turned into 30 pages. To make this information more consumable, we broke the article into two parts, with this first installment describing the underlying concepts and some things that should be in

place *before* you start a tuning exercise. The second installment will show how you use system commands and SMF data to identify good candidates for your tuning activities and then determine the savings you achieved. We hope you find them helpful.

Background

Terminology

We seem to have a gift in the mainframe world for coming up with confusing terminology and acronyms. So, to avoid confusing readers, especially those who might be relatively new to this platform, there is one set of acronyms that we need to clarify up front:

- PDS's - Plural of PDS (Partitioned Data Set). That is, two or more instances of *traditional* Partitioned Data Sets.
- PDSEs - Plural of PDSE (Partitioned Data Set Extended).

In this article, we will use the term 'partitioned data set' to refer generically to either PDS's or PDSEs. If the topic of the text is relevant to only one or other of them, we will explicitly say 'PDS' or 'PDSE'.

Another term you will see a lot is 'SMSPDSE(1)'. Depending on how you set up SMS, you *might* have *two* address spaces that manage PDSE data sets - SMSPDSE and SMSPDSE**1**. There are many commands and parameters that apply to both address spaces, so we will follow the IBM convention of using 'SMSPDSE(1)' when referring to something that applies to both address spaces or 'PDSE(1)' when referring to an option that might be PDSE or PDSE1. For example, there are two IGDSMSxx parameters called `PDSE_LRUTIME` and `PDSE1_LRUTIME` - so rather than constantly saying "`PDSE_LRUTIME` and `PDSE1_LRUTIME`" we will simply say "`PDSE(1)_LRUTIME`".

This article assumes that the reader is familiar with PDSEs, and how they differ from traditional partitioned data sets. It is not an introduction to PDSEs, but rather a refresher on the performance aspects of partitioned data sets, and especially PDSEs. There have been a number of changes over recent years that impact this topic, so some existing documentation might not reflect the latest information.

An additional motivation for an article on this topic is the move to an accumulated-MSU-based model for software pricing (Tailored Fit Pricing). Moving from an R4HA model to an accumulated CPU time model means that *anything* that you can do to make your system run more efficiently will have a positive impact on your bill. We are seeing clients doubling or even tripling the amount of memory when upgrading their CPCs, so why not use a tiny portion of that to improve the efficiency of your partitioned data sets?

I'm sure that some of you are thinking that, compared to the tens of millions of I/Os to your database data sets, the number of I/Os that are issued to PDS or PDSE data sets is trivial. And you are probably correct. However, all those 'trivial' savings can make a real difference when you add them up. And if an investment of a few days of your time can result in the system running more efficiently *every day* thereafter, why would you *not* do that?

Real World Example

Some time ago we helped a client with a system tuning exercise. They had one job that was running so long that it was starting to impact their batch schedule. The job was using the IBM Workload Scheduler Workload Automation Programming Language (WAPL). Analysis of the SMF type 30 records for the job showed that it was doing over 1,000,000 I/Os to the IWS load library. To reduce the number of I/Os, they added that load library to the LNKLST which had the positive effect of having the load library and its directory entries managed by LLA. This resulted in a 99% reduction in the number of I/Os issued by the job, a 53% reduction in elapsed time, and a 5% reduction in CPU time.

Will every job get that much benefit? Of course not, but a small number of improvements like this could make you very popular with the owners of the affected jobs.

We will start with a brief summary of two MVS functions that are commonly used to improve performance for users of traditional partitioned data sets, LLA and VLF, and then dive into the latest information and considerations for PDSE data sets.

Library LookAside and Virtual Lookaside Facility

Library LookAside (LLA) and Virtual Lookaside Facility (VLF) have been around for a *long* time. I still had brown hair and a motorcycle when they came out! So, we are not going to give you a lesson about LLA and VLF. However, there *are* some common misunderstandings that we would like to address.

But first, briefly, what do they do? LLA:

- Maintains a cache containing the *directory entries* for 'selected' partitioned data sets (both PDS and PDSEs).
- Maintains statistics that it uses to decide which *load modules* or *program objects* should be cached in VLF.

VLF, on the other hand, is a generalized caching capability that provides services to *multiple* exploiters. LLA is just one of those exploiters (but only for load modules or program objects - LLA's *directory* cache is in LLA's own address space). Other VLF exploiters you might be familiar with are TSO/E (IKJEXEC, for Clists and Rexx execs) and CATALOG (IGGCAS).

Let's get into a little more detail about LLA. LLA can manage:

- LNKLST data sets.
- Non-LNKLST PDS's that contain load modules.
- The *directories* of other PDS's. These could contain ISPF panels or JCL procedures or parameters, or anything at all. Note that if used for a non-load library PDS, the benefit of LLA is limited to directory I/O; *the members are not cached*. Also, take note of the comments later in this section about the use of FREEZE for these PDS's.

Note: This capability is not mentioned in the 'Improving module fetch performance with LLA' section of the *z/OS Initialization and Tuning Guide*. However, in the *DFSMS Using Data Sets* manual you can see that the BLDL and DESERV services used to search for PDS and PDSE members have an option to bypass LLA search which defaults to NO. This means that unless the program explicitly states ALL, PDS searches *will* go through LLA first.

We think this is interesting to know, and something that some readers might be unaware of. Also, there may be products or components that use that bypass option, making adding the relevant libraries to LLA useless.

- PDSE data sets that contain program objects. Note that the *only* PDSEs that can be managed by LLA are those that contain program objects. If you add a PDSE that contains anything *other* than program objects to LLA you will receive a `CSV242I INVALID DATA SET ORGANIZATION FOR LLA DSN:` message, followed by LLA ending with a S023 abend.

If you start the LLA address space and do not point it at *any* CSVLLAxx member[5], it maintains the directory entries for all LNKLST data sets.

You can use the CSVLLAxx member(s) of Parmlib to identify additional, non-LNKLST, PDS's whose directories you want LLA to cache. If the data set is in LNKLST or has been included on the FREEZE statements in the CSVLLAxx member, BLDL or DESERV requests to get information from the data set's directory will use the LLA directory cache.

LLA can be used to cache the directories of all types of PDS's - it is not limited to load libraries.

In addition to caching the *directories* of the LLA-managed data sets, LLA is able to use VLF to cache *program objects* or *load modules*. The exact logic that LLA uses is not published, but it is based on information such as module size, frequency of fetches per module (fetch count), and the time required to fetch a particular module. There are two LLA exits, CSVLLIX1 and CSVLLIX2, that can be used to influence which objects

[5] Note that LLA does NOT read the CSVLLA00 member unless you explicitly identify that member on the **S LLA** command.

are passed to VLF by LLA. As you will see later, this is a more sophisticated implementation than that employed by the SMSPDSE and SMSPDSE1 address spaces when deciding which PDSE members should be cached in their respective Hiperspaces. This is one of the reasons that we recommend that PDSE program object data sets are included in LLA - but more about that later.

For both PDS's and program object PDSEs, the LLA directory cache is only used for data sets that are in LNKLST or are defined as FREEZE in the CSVLLAxx member. You can use the **D LLA** command to determine the current FREEZE/NOFREEZE status for the data sets defined to LLA, as shown in Example 1.

Example 1 - Output from D LLA Command

```
D LLA
CSV600I 13.38.51 LLA DISPLAY 129
EXITS: CSVLLIX1 - ON   CSVLLIX2 - ON
VLF: ACTIVE  GET LIB ENQ: YES  SEARCH FAIL COUNT: 0
LNKLST SET: LNKLST00
52 LIBRARY ENTRIES FOLLOW
ENTRY   L F R P  LIBRARY NAME
    1   L        GDDM.SADMMOD
    2   L        SYS1.CSSLIB
    3   L        SYS1.SBDTLIB
    4   L     P  EQAE20.SEQAMOD
   48   L        ISF.SISFLOAD
   49   L        SYS1.SBDTCMD
   50     F      SYS1.SEDGMENU
   51   L        CSF.SCSFMOD0
   52   L        SYS1.DGTLLIB
```

An 'F' in the 'F' column indicates a data set that is defined to LLA with FREEZE - in this example, the SYS1.SEDGMENU data set was defined as FREEZE in the CSVLLAxx member. Note that LNKLST data sets (those with an 'L' in the 'L' column) are always treated as FREEZE when used as part of the LNKLST[6], even though they don't have an F in the F column.

If you want to compare the list of LLA-managed data sets to the data sets that are in LNKLST, the easiest way is to use the L column in the output from this command - the subset of data sets that contain an 'L' in that column are the LNKLST data sets. (The 'P' column in the **D LLA** output indicates data sets that are PDSEs.)

If a non-LNKLST data set is defined to LLA, but is not included on a FREEZE statement in CSVLLAxx, LLA will not load that data set's directory into its cache. So, you might wonder why would you define a data set to LLA and not include it on the FREEZE statement[7]?

[6] LNKLST data sets are not treated as FREEZE if used (for example) in a JOBLIB/STEPLIB/TASKLIB concatenation.

[7] Note that the default for non-LNKLST data sets is NOFREEZE.

Well, you could have a load library that is updated frequently enough that you don't want to have to deal with switching the data set back and forth between FREEZE and NOFREEZE, or having to tell LLA to refresh that data set every time it is updated. However, you would still like it to benefit from having its load modules cached in VLF. In that case, you might want to include that data set in LLA as NOFREEZE. This means that LLA can cache the programs from that data set in VLF, but it avoids the management complexity associated with FREEZE data sets. Note that this only applies to PDS's that contain load modules, or PDSEs that contain program objects. I don't believe there is any benefit in including other types of PDS's in LLA if you are going to define them as NOFREEZE.

Managing and Monitoring LLA and VLF

There are system commands, a Health Check, a new function in SDSF in z/OS 3.1 (based on a no-longer-available tool called the Module Fetch Monitory), and SMF records that you can use to gain insight into what LLA and VLF are doing. However, that information is also pertinent when you are looking at optimizing PDSE performance, so we will cover them in the 'Monitoring and Managing Partitioned Data Set Performance' section in the next article in this series.

You can find more information about LLA in the section titled 'Improving module fetch performance with LLA' in the *z/OS MVS Initialization and Tuning Guide*. For more information about VLF, refer to the 'The virtual lookaside facility (VLF)' chapter in the *z/OS MVS Authorized Assembler Services Guide*.

PDSEs

Partitioned Data Set Extended (PDSE) data sets are described in 'Chapter 27 Processing a partitioned data set extended (PDSE)' in *z/OS DFSMS Using Data Sets*, SC23-6855 and in IBM Redbook *Partitioned Data Set Extended Usage Guide*, SG24-6106. If you are not familiar with PDSEs, we recommend that you read either or both of those documents. In *this* article we will focus on just one aspect of PDSEs - using them to optimize the performance of your partitioned data sets.

Note: Before we proceed, we just wanted to point out to readers who might not be familiar with PDSEs that, unlike PDS's which do not have any caching mechanism of their own, DFSMS *does* provide separate caching capabilities for each of PDSE directories and PDSE members.

Prerequisites

There are some settings in SMS that, strictly speaking, are not prerequisites for the functions we describe in this article. However, we *strongly* recommend that you ensure they are in place before proceeding, so we are covering them in this first article. Hopefully, you will have an opportunity to make sure all these pre-reqs are in place by the time the next Tuning Letter issue is ready for your review.

If you decide to convert some of your current PDS's over to PDSE format, you should do all you can to ensure that the IGDSMSxx settings that optimize availability, recoverability, and performance are in place *prior* to that conversion.

SMSPDSE1 Address Space

The *default* PDSE configuration is to have just the one system address space to manage all PDSEs - that address space is called SMSPDSE. That address space is not restartable. However, you can *optionally* request that a second, restartable, PDSE address space called SMSPDSE1 is started.

If you specify `PDSE_RESTARTABLE_AS(YES)` in your IGDSMSxx member of Parmlib, the SMSPDSE address space will be dedicated to handling I/Os to PDSEs that are accessed via LNKLST; all other PDSE accesses will be managed by the SMSPDSE1 address space.

There is also a potential availability benefit from enabling the SMSPDSE1 address space. Like all software, it is possible for problems to occur in the PDSE address space. If those problems occur in the restartable SMSPDSE1 address space, you can often address the problem by restarting just that address space. However, if the same problem occurred in the (non-restartable) SMSPDSE address space, the only way to resolve it would be to IPL the system.

Note: While the SMSPDSE1 address space *is* restartable, there are some limitations and restrictions - see the 'Considerations for restarting the SMSPDSE1 address space' section in the *z/OS DFSMS DFP Diagnosis Guide* for more information.

There is also a flexibility benefit. If you do not enable the SMSPDSE1 address space, some changes to IGDSMSxx require an IPL. Because the SMSPDSE1 address can be restarted, changes to some of the PDSE1* parameters in IGDSMSxx[8] can be activated dynamically. The LNKLST tends to be relatively stable, so if you want to make changes to the IGDSMSxx member, the chances are that those changes will be related to *non*-LNKLST PDSEs. The use of SMSPDSE1 can therefore potentially help you implement changes without requiring an IPL.

[8] Note that some changes to the SMSPDSE and even the SMSPDSE1 address space parameters might still require an IPL.

For these reasons, we *highly* recommend that you enable the SMSPDSE1 address space by specifying `PDSE_RESTARTABLE_AS(YES)` in your IGDSMSxx member of Parmlib.

Enabling the SMSPDSE1 address space requires an IPL.

The easiest way to check if this has been done already is to issue the **`D A,SMSPDSE*`** command - if the response shows just a single address space, then `PDSE_RESTARTABLE_AS(YES)` was *not* used the last time the system was IPLed. Unfortunately, this setting can only be put into effect by IPLing the system. Note that it IS possible to have this enabled on one system while other members of the sysplex are still using `PDSE_RESTARTABLE_AS(NO)`- in other words, this can be implemented using rolling IPLs.

Recommendation: Many of the parameters in the IGDSMSxx member can be changed dynamically using one of two methods:

- **`SET SMS=xx`** command - This command tells SMS to read the referenced member of Parmlib, check the syntax of all statements in that member, and (where possible) activate any changes from that member.
- **`SETSMS parameter(new_value)`** - This command changes only the parameter specified on the command. The IGDSMSxx member is not updated and the change is only in effect until the next time the system is IPLed.

Generally speaking, we recommend that you use the first option. This syntax checks the IGDSMSxx member while your system is still up and running, so any typos can be easily fixed. It also ensures that your change won't be accidentally backed out when the system is next IPLed.

However, according to the z/OS MVS Initialization and Tuning Reference:

> "*Some parameters in the IGDSMSxx parmlib member are read at IPL time, but do not have any effect unless it is the first time the parameters are specified for a new configuration. As indicated in the section on changing IGDSMSxx parameters to support the coupling facility in z/OS DFSMSdfp Storage Administration, you must issue the SETSMS command for any changes to take effect. In particular, the values used by VSAM RLS for the following keywords are remembered from last initialization, but not overridden by the IGDSMSxx specifications at IPL time unless the SETSMS command is issued*."

Is this confusing and inconsistent with how most components are controlled using the **`SET mbr=xx`** and **`SETmbr parm=value`** commands? Yes. But don't shoot us, we are only the messengers.

Warning: Changes to some of the PDSE1* parameters in the IGDSMSxx member are only *implemented* when the SMSPDSE1 address space is next restarted, or when the system is next IPLed. However, the output from the **D SMS,OPTIONS** command shows the current setting *of the parameter* - this does *not* necessarily reflect the value that is currently being used by the system.

For example, you could issue a **SETSMS PDSE1__HSP_SIZE(100)** command to change the size of the SMSPDSE1 Hiperspace cache. The command will be accepted. And if you then issue a **D SMS,OPTIONS** command it will tell you that PDSE1_HSP_SIZE is set to 100. However, in reality, the system will continue to use the previous setting until SMSPDSE1 is restarted. This situation also applies if you update the IGDSMSxx member and then issue a **SET SMS=xx** command to 'activate' your change.

At the time of writing, there are 83 IGDSMSxx parameters, and we simply don't have time to test every one to see if the output from the **D SMS,OPTIONS** command accurately reflects the current state of that function or option. Therefore, to avoid confusion, we *highly* recommend scheduling your changes for a time when you can restart SMSPDSE1 (or IPL, if that is what is required) as soon as possible after you issue the command to change the SMS parameters.

Extended Sharing mode is a prerequisite for enabling the restartable SMSPDSE1 address space.

Note that the restartable address space will only be started if your system is running in Extended Sharing mode (covered in the next section down). If you IPL with PDSE_RESTARTABLE_AS set to YES, but with PDSESHARING either not specified or set to NORMAL, the system will start, but you will receive message `IGW035I SMSPDSE1 IS NOT ENABLED, SMS=00`. However, that message is easily missed during the flood of messages that are issued while the system is initializing.

Don't believe everything that the D SMS,OPTIONS command tells you.

To further confuse matters, if you then issue a **D SMS,OPTIONS** command, the response will tell you that PDSE_RESTARTABLE_AS is set to YES, but, if you issue a **D A,SMSPDSE*** command, the response will show that only the SMSPDSE address space is running. During our testing of the various PDSE functions, we found the mismatches between the response from the **D SMS,OPTIONS** command, and some of the options that the SMSPDSE(1) address spaces *were actually using*, to be quite confusing.

Recommendation: We are not aware of any good reason for *not* enabling the SMSPDSE1 address space. Therefore, all the examples and descriptions in the remainder of this article assume that it *is* enabled. If there are any exceptions, we will explicitly point them out.

Extended Sharing Mode

There are two sharing modes for PDSEs: NORMAL or EXTENDED. The default is NORMAL, but the *recommended* mode is EXTENDED. EXTENDED mode has been available for many years now, and most of our clients run in EXTENDED mode. However, it has some restrictions, the most important of which is that all systems that will share a PDSE data set must be in the same GRSplex *and* in the same sysplex (because the SMSPDSE(1) address spaces in the different systems coordinate their activity using XCF). You really should not be sharing any data set between a system that is in a sysplex and another system that is *not* in the same sysplex anyway, so hopefully, this inability to share PDSEs outside the sysplex will not be an issue for you.

For some reason, I previously, *incorrectly*, thought that a sysplex IPL is required to move from NORMAL to EXTENDED sharing mode. It is true that all systems in the sysplex must be using the same sharing mode, however, the 'Specifying Extended PDSE Sharing in a Multiple-System Environment' section in the *z/OS DFSMS Using Data Sets* manual describes a way to move from NORMAL to EXTENDED sharing using rolling IPLs.

I'm not sure why anyone would want to do it, but if you are currently using EXTENDED sharing and you want to revert to NORMAL mode, that *does* require a sysplex IPL. You can find more information about that in the 'Specifying Normal PDSE Sharing in a Multiple-System Environment' section in the *z/OS DFSMS Using Data Sets* manual.

PDSE(1)_BUFFER_BEYOND_CLOSE

The default mode of operation for PDSE caching is that the directory blocks (and members, for data sets that are eligible for member caching) are removed from the caches when the last user of that data set on that system closes it. If you have PDSEs that are frequently opened and closed, you can avoid having to re-read the directory on each open by specifying the PDSE(1)_BUFFER_BEYOND_CLOSE(YES) options in the IGDSMSxx member of Parmlib. This will cause the members and directory entries to remain in their respective caches until they are flushed using the normal Least Recently Used mechanism.

To ensure the integrity of any directory or member blocks in its buffers, BUFFER_BEYOND_CLOSE uses XCF to inform other systems when a block is updated. The use of XCF for this communication means that all the systems that are sharing the data set *must* be in the same sysplex *and* using EXTENDED mode sharing.

BUFFER_BEYOND_CLOSE requires EXTENDED sharing mode.

Note that you *can* specify BUFFER_BEYOND_CLOSE(YES) in your IGDSMSxx member if PDSESHARING is set to NORMAL, and if you issue the **`D SMS,OPTIONS`** command it will report that BUFFER_BEYOND_CLOSE is enabled - however that information is misleading.

Even though it might show that it is enabled, the SMSPDSE(1) address spaces will ignore that setting until the sysplex is running in EXTENDED sharing mode.

PDSE(1)_HSP_SIZE

PDSE *member* caching is *disabled* by default. You enable it by specifying a non-zero value for the Hiperspace size on the PDSE(1)_HSP_SIZE parameters in the IGDSMSxx member. If the SMSPDSE1 address space is enabled, the SMSPDSE Hiperspace will only be used for program objects accessed via LNKLST. The SMSPDSE1 Hiperspace will be used for all other PDSE data sets that are eligible for caching, *and* for non-LNKLST I/Os to LNKLST data sets.

Changes to the size of the SMSPDSE Hiperspace require an IPL, so you should try various values on a test system first to determine a reasonable size for your environment. Changes to the SMSPDSE1 Hiperspace size can be implemented dynamically, but they do require a restart of the SMSPDSE1 address space.

If you have not changed the default values, then you are not using member caching today. In that case, just about any value greater than zero should deliver an improvement. A good place to start might be to decide how much memory you are willing to use for caching your PDSEs and use that as your initial values. The SMSPDSE(1) address spaces only allocate as much memory as they require, so if you specify a HSP_SIZE of 2000 MB[9], but only need 100 MB, only 100 MB will be allocated. Unfortunately, there is no command to display the current size of the Hiperspace or how full it is[10].

PDSE(1)_LRUTIME

The LRUTIME parameter, combined with the LRUCYCLES parameter, is used by the SMSPDSE(1) address spaces to decide how frequently to clean up the member cache Hiperspaces. The default LRUTIME is 60 seconds and the default LRUCYCLES value is 15, meaning that SMSPDSE(1) will clean up their caches every 15 minutes.

IBM doesn't recommend changing those values, and I've never seen a customer who found it necessary to change them. So, why are we mentioning them? Because of a tip suggested in section '3.1.4 PARMLIB IGDSMSxx parameters for SMSPDSE1 support' of the IBM *PDSE Usage* Redbook (SG24-6106).

Both of the SMSPDSE(1) address spaces create SMF Type 42.1 records that contain system- and storage-class-level information about the use of their directory and member caches. This is helpful information. However, for some reason, the records don't contain any

[9] The description of the PDSE(1)_HSP_SIZE parm in the z/OS MVS Initialization and Tuning Reference infers that the maximum supported value is 3 characters long. While we doubt that anyone will require a cache larger than 999 MB, the parm does actually support a value up to 2047.

[10] You can use the **`D SMS,PDSE(1),VSTOR`** command the determine the maximum size and used memory in the directory dataspace, but there is no corresponding command for the member cache.

indication of which of the SMSPDSE(1) address space they came from. If you are going to use that information to fine-tune the size of the address space's Hiperspace, it would be nice to know which address space's statistics you are looking at!

The 42.1 records are quite compact, but one piece of information they *do* contain is the LRUTIME value for the address space that created the record. The Redbook team suggested using slightly different LRUTIME values for SMSPDSE and SMSPDSE1. You can then use the SMF42LRU field in the SMF record to identify which SMSPDSE* address space created the record.

Consider changing the LRUTIME value, as proposed in the PDSE Usage Redbook.

SMF Intervals

The default interval for the creation of SMF 42.1 records is 3600 seconds. For normal, steady-state operation that might be OK. However, while you are testing different IGDSMSxx and SMS storage class settings, it is helpful to have both more granularity, and also to not have to wait so long to see the impact of your changes. Therefore, there are two IGDSMSxx parameters that you should check.

The interval between creation of the SMF 42.1 records is controlled by the PDSE(1)_BMFTIME parm. The default value is 3600 seconds, but we recommend that you change that to match the length of your SMF interval. Unfortunately, there is no option to tell it to automatically use the same interval as SMF - you need to specify a number of seconds. But the good news is that this can be changed dynamically, without having to restart SMSPDSE(1) or IPL the system.

We recommend changing BMFTIME to match your SMF interval.

The other, related, parameter, is SMF_TIME. This is *supposed* to control whether the creation of certain SMS SMF type 42 records (including the subtype 1, 2, 15, 16, 17, 18, 19) is synchronized with the SMF interval. The default value for this is YES, meaning that the records should be synchronized to the SMF interval. However, in our tests, the Type 42.1 records were consistently created based on the time the corresponding SMSPDSE(1) address space finished initializing, while the SMF interval was synchronized on the 00 minute.

That concludes *this* installment of the *Optimizing PDS and PDSE Performance* article. Hopefully this article has been valuable in helping you better understand the roles and capabilities of LLA and VLF, and the settings in your IGDSMSxx member to help you achieve optimal performance and availability.

References

Even though we only covered one relatively small aspect of z/OS in this article, you can see that this is quite a complex topic. You can find additional information in the following documents:

- IBM Redbook *Partitioned Data Set Extended Usage Guide*, SG24-6106. This is an excellent source of information. However, it is 15 years old and *some* details have changed since it was written - for example, it doesn't mention PDSE V2.
- IBM Redbook *z/OS V1R8 DFSMS Technical Update*, SG24-7435, particularly section 3.9 'PDSE buffer management statistics'.
- IBM Manual *z/OS DFSMS Using Data Sets*, SC23-6855.
- IBM Manual *z/OS MVS Initialization and Tuning Guide*, SA23-1379.
- IBM paper *An Overview of Hiperspace Caching for PDSE*, by **Trevor Geisler**.
- IBM paper *PDSE Data Set Sharing Basics*.
- SHARE in New Orleans 2023 Session 31030, '*PDSE Pitstop: Performance, Problem Analysis and Pheatures*', by **Tabari Alexander** and Trevor Geisler**.**
- 'Optimizing Efficiency of COBOL V5 Load Processing' article in *Tuning Letter 2015 No. 1*.
- Blog post by **Colin Paice** titled '*Avoiding I/O by caching your PDSEs (It might not be worth it)*.

Summary

While PDS and PDSE data sets might not get the attention that is showered on the large database managers, they still play an important role in most z/OS sites. I would challenge you to name *any* product that doesn't use any form of partitioned data set. As a result, there are many potential candidates out there that are ripe for a little tuning.

The traditional performance optimization capabilities for PDS data sets, LLA and VLF, are still out there, and still just as valuable, but we frequently find that they are not being exploited to their maximum potential.

PDSE data sets are new*er*, but they are not new. But we find that they frequently suffer from the same lack of Tender Loving Care that afflicts the PDS data sets. This is a shame, because a small investment of time can deliver real performance and availability benefits.

In this installment we introduced you to the basic concepts and we listed the IGDSMSxx settings that we recommend. Hopefully you have most of these in place already. If not, you can take this opportunity to identify what changes are required and include them in your planned system outages over the next couple of months.

We also strongly encourage you to take the information provided in this article about anomalies in the PDSE documentation or in the way PDSE behaves and play around with it yourself in a test system. Ensuring that you clearly understand how this works will save you many hours of head-scratching and frustration when you start implementing these changes in your production environment.

With a solid base under you, the next article in this series will show you how PDSE directory and member caching work, the commands and SMF data you can use to determine the impact of your changes, and show you how to select the best candidates for your partitioned data set optimization project.

We hope you found this article to be helpful. Please let us know if you have any additional tips or experiences that you would like us to pass along to your fellow readers.