

MP16: Capacity Planning and Tuning for IBM MQ for z/OS

May 2024

**IBM MQ Performance
IBM UK Laboratories
Hursley Park
Winchester
Hampshire
SO21 2JN**

Take Note!

Before using this report, please be sure to read the paragraphs on “disclaimers”, “warranty and liability exclusion”, “errors and omissions” and other general information paragraphs in the “Notices” section below.

Thirteenth edition, May 2024. This edition applies to IBM MQ for z/OS version 9.3.x (and to all subsequent releases and modifications until otherwise indicated in new editions).

© Copyright International Business Machines Corporation 2024.
All rights reserved.

Note to U.S. Government Users Documentation related to restricted rights. Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule contract with IBM Corp.

Notices

DISCLAIMERS

The performance data contained in this report were measured in a controlled environment. Results obtained in other environments may vary significantly.

You should not assume that the information contained in this report has been submitted to any formal testing by IBM.

Any use of this information and implementation of any of the techniques are the responsibility of the licensed user. Much depends on the ability of the licensed user to evaluate the data and to project the results into their own operational environment.

WARRANTY AND LIABILITY EXCLUSION

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

In Germany and Austria, notwithstanding the above exclusions, IBM's warranty and liability are governed only by the respective terms applicable for Germany and Austria in the corresponding IBM program license agreement(s).

ERRORS AND OMISSIONS

The information set forth in this report could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; any such change will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time and without notice.

INTENDED AUDIENCE

This report is intended for Architects, Systems Programmers, Analysts and Programmers wanting to understand the performance characteristics of **IBM MQ for z/OS V9.3 and earlier releases**. The information is not intended as the specification of any programming interfaces that are provided by IBM MQ. Full descriptions of the WebSphere MQ facilities are available in the product publications. It is assumed that the reader is familiar with the concepts and operation of IBM MQ.

Prior to IBM MQ for z/OS V8.0, the product was known as WebSphere MQ and there are instances where these names may be interchanged.

LOCAL AVAILABILITY

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which IBM operates. Consult your local IBM representative for information on the products and services currently available in your area.

ALTERNATIVE PRODUCTS AND SERVICES

Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

USE OF INFORMATION PROVIDED BY YOU

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

TRADEMARKS and SERVICE MARKS

The following terms, used in this publication, are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both:

- IBM®
- z/OS®
- zSeries®
- IBM Z®
- zEnterprise®
- MQSeries®
- CICS®
- Db2 for z/OS®
- IMS™
- MVS™
- z10™
- zEC12™
- z13™
- z14™
- z15™
- z16™
- FICON®
- WebSphere®
- IBM MQ®

Other company, product and service names may be trademarks or service marks of others.

EXPORT REGULATIONS

You agree to comply with all applicable export and import laws and regulations.

Summary of Amendments

Date	Changes
2024-May	<p>Version 2.12 include updates for:</p> <ul style="list-style-type: none"> • Updated message selector performance • TLS start channel performance • Channel initiator task storage usage • Remove references to out-of-support MQ releases • Queue manager restart time with deep indexed queues
2023-May	<p>Version 2.11 include updates for:</p> <ul style="list-style-type: none"> • Updated Common MSGID or CORRELID with deep shared queues • Updated Maximum message rate for private and shared queues • Updated Shared Queue persistent message throughput after 63KB transition for Db2 v13. • Using AT-TLS to encrypt data flowing over MQ channels - updated for 10Gb network on z16. • Start/stop channel performance on z16.
2022-August-November	<p>Version 2.10 include updates for:</p> <ul style="list-style-type: none"> • MQ for z/OS 9.3 - ECSA usage, queue statistics • MAXARCH - clarification of maximum size of recoverable data • Impact of updating DASD, including maximum request/reply throughput of persistent messaging on DS8900F • CFCC 25 on IBM z16 • zHyperWrite on IBM z16 and DS8950F
2021-September	<p>Version 2.9 include updates for MQ channel performance:</p> <ul style="list-style-type: none"> • Channel start and stop performance • Using AT-TLS to encrypt data flowing over MQ channels - updated for TLS 1.3 • Updates to first-open, last-close effects
2020-October	<p>Version 2.8 include updates for MQ version 9.2:</p> <ul style="list-style-type: none"> • Dataset encryption of page set, log and SMDS • zHyperwrite support for active log • TLS 1.3 cipher support on MQ channels

Table of contents

1 Queue Manager	1
Queue manager attributes	1
Log data set definition	1
Should your archive logs reside on tape or DASD?	1
How to minimize recovery time	2
Should your installation use single or dual logging?	2
How large can your active logs be?	3
Striped logs	4
Striped archive logs	4
8-byte log RBA	5
How much log space does my message use?	5
What is my logging rate?	8
How much log space do I need when backing up a CF structure?	9
How can we estimate the required log data rate for a system?	9
Page sets	11
Page set usage	11
Size of page sets for a given number of messages	11
Number of page sets	13
Recovering page sets	13
How often should a page set be backed up?	13
Why WebSphere MQ for z/OS changed how it manages small messages in V7.0.1	15
Buffer pools	19
Buffer pool default sizes	19
Buffer pool usage	19
Using buffers allocated in 64-bit storage	21
Page fixed buffers	21
Why not page fix by default?	21
The effect of buffer pool size on restart time	21
Deferred Write Process	21
What changed in version 8.0?	22
Oversized buffer pools	22
How many DWP tasks are there?	22
How much data could DWP write at checkpoint?	22
What impact is there when DWP writes large amounts of data?	23
Recovery	24
Restart	25
How long will my system take to restart after a failure?	25
What happens during a checkpoint	27
What happens during the recovery phase of restart	28
How long will each phase of the recovery take?	30
What happens during the recovery phase of restart when in a QSG	39

Worked example of restart times	40
Tuning	42
Performance implications of very large messages	42
Queue Manager attribute LOGLOAD	42
What is LOGLOAD?	42
What settings are valid for LOGLOAD?	42
What is an appropriate value for LOGLOAD?	42
When might a lower LOGLOAD be appropriate?	42
What happens at checkpoint	43
Impact of LOGLOAD on workload	43
Impact of LOGLOAD on log shunting	45
Impact of LOGLOAD on restart	45
Use of MQ Utilities	46
IBM MQ Utilities: CSQUTIL	48
Queue Manager Trace	50
Accounting Trace Costs	50
Storage Usage	51
Who pays for data collection?	51
Who pays for writing to SMF?	51
How much data is written?	51
Statistics Trace Costs	53
Global Trace Costs	54
Performance / Scalability	56
Maximum throughput using persistent messages	56
What factors affect persistent message throughput ?	56
Application syncpoint specifics	56
Message size and number of messages per commit	57
Indexed Queues	58
Indexed queue considerations	58
Private indexed queue rebuild at restart	58
How long will it take to restart a queue manager with deep indexed local queues	58
The effect of a single deep indexed queue upon Queue Manager restart	59
The effect of a multiple deep indexed queues upon Queue Manager restart	60
Queue manager initiated expiry processing	60
Queue manager security	62
How much storage is used?	62
The environment being measured	62
The data	64
What can we gather from the chart?	64
Virtual storage usage	65
Object sizes	66
Page set 0 usage	66
Virtual storage usage by object type	67
Initial CSA (and ECSA) usage	68
CSA usage per connection	68
Buffer Pool Usage	68
Storage for security information	68
Impact of number of objects defined	69
Use of indexed queues	69
Object handles	69
Number of pages in use for internal locks	69
Shared queue	70
Using BACKUP CFSTRUCT command	70
Clustering	70

2 Coupling Facility	71
CF link type and shared queue performance	71
How many CF structures should be defined?	72
What size CF structures should be defined?	72
CSQ_ADMIN	73
How large does my admin structure need to be?	74
Application structures	75
How many messages fit in a particular CF application structure size?	76
CF at CFCC levels 17 and later	77
Sizing structures at CFLEVEL(5)	77
Increasing the maximum number of messages within a structure	79
Use of system initiated alter processing	80
User initiated alter processing	80
How often should CF structures be backed up?	80
Backup CFSTRUCT limits	82
Administration only queue manager	83
When should CF list structure duplexing be used?	84
How does use of duplexed CF structures affect performance of MQ?	85
CPU costs	85
Throughput	85
CF Utilization (CF CPU)	85
Environment used for comparing Simplex versus Duplex CF structures	86
Duplexing the CSQ_ADMIN structure	86
Duplexing an application structure	87
Non persistent shared queue message availability	87
Coupling Facility	88
What is the impact of having insufficient CPU in the Coupling Facility?	88
When do I need to add more engines to my Coupling Facility?	88
What type of engine should be used in my Coupling Facility?	88
CF Level 19 - Thin Interrupts	88
CF Level 25 - Thin Interrupts	88
Why do I see many re-drives in the statistics report?	89
What is a re-drive?	89
Why do I see many re-drives in the statistics report?	89
Effect of re-drives on performance	90
Batch delete of messages with small structures - CFLEVEL(4) and lower	91
Shared Message Data Sets - CFLEVEL(5)	91
Tuning SMDS	91
DSBUFS	91
DSBLOCK	94
CFLEVEL(5) and small messages	96
Who pays for messages stored on Shared Message Data Sets?	96
Db2	97
Db2 universal table space support	97
Is Db2 tuning important?	98
Why does IBM MQ produce more Db2 rollbacks than I expect?	98
Shared queue messages > 63KB	98
Shared queue persistent message throughput after 63KB transition	101
Shared queue persistent message request/reply CPU costs	104
Shared queue persistent message request/reply CF costs	104
Storage Class Memory (SCM)	106
Using SCM with IBM MQ	106
Impact of SCM on Coupling Facility capacity	106
How much SCM is available?	108

How do I know I am using SCM and how much?	108
ALLOWAUTOALT(YES) usage with SCM	108
Useful guidelines:	108
Impact of SCM on Application performance	109
Non-Sequential gets from deep shared queue	110
RMF data	110
Example use cases for IBM MQ with SCM	111
Capacity – CFLEVEL(4 and less) – no offload available - “Improved Performance”	112
Capacity – CFLEVEL(5) Offload - “Emergency Storage”	112
Capacity – CFLEVEL(5) – no offload - “Improved Performance”	112
Performance / Scalability	113
Does the CF Structure attribute “CFLEVEL” affect performance?	113
The impact on MQ requests of the CURDEPTH 0 to 1 transition	113
When would I need to use more than one structure?	114
When do I need to add more Queue Managers to my QSG?	114
What is the impact of having Queue Managers active in a QSG but doing no work?	114
What is a good configuration for my shared queues?	114
Shared queue persistent messages	115
Shared queue performance affecting factors	115
CFRM Attributes	116
3 Channel Initiator	117
What is the capacity of my channel initiator task?	117
Channel initiator task storage usage	118
What limits the maximum number of channels?	118
How many channels can a channel initiator support?	118
How many SVRCONN channels can a channel initiator support?	119
Does SSL make a difference to the number of channels I can run?	119
Channel initiator buffer pools	120
What happens when the channel initiator runs out of storage?	121
Channel Initiator Scavenger Task	121
Defining channel initiator - CHINIT parameters	122
CHIADAPS	122
CHIDISPS and MAXCHL	123
Checking the OMVS Environment	124
Effect of Changing CHIDISPS	125
Tuning Channels	126
Channel option BATCHHB	126
Channel option BATCHINT	126
Channel option BATCHLIM	126
Channel option BATCHSZ	126
Channel option COMPHDR	126
Channel option COMPMMSG	127
Channel option DISCINT	129
Channel option HBINT	129
Channel option KAIT	129
Channel option MONCHL	130
Channel option NPMSPEED	130
SVRCONN channel option SHARECNV	130
Tuning channels - BATCHSZ, BATCHINT, and NPMSPEED	131
How batching is implemented	131
Setting NPMSPEED	132
Determine achieved batch size using MONCHL attribute	133

Setting BATCHSZ and BATCHINT	134
Channel Initiator Trace	136
Why would I use channels with shared conversations?	137
Performance / Scalability	138
Channel start/stop rates and costs	138
TLS channel start costs	139
Factors affecting channel throughput and cost	140
SSL and TLS	141
When do you pay for encryption?	141
How can I reduce the cost?	142
Will using cryptographic co-processors reduce cost?	143
TLS 1.3 cipher support	143
Why use TLS 1.3?	144
Deprecated CipherSpecs	144
Starting TLS channels using aliases	145
Stopping TLS channels	145
Secret key negotiation costs	146
Cost of encryption using TLS ciphers	147
Can I influence which cipher is chosen?	148
SSLTASKS	149
How many do I need?	149
Why not have too many?	149
Why not have too few?	149
SSLTASK statistics	149
SSL channel footprint	150
SSL over cluster channels	150
SSL over shared channels	150
Using AT-TLS to encrypt data flowing over IBM MQ channels	151
Who pays for AT-TLS	151
Limitations	151
Performance comparison	151
Is the reduced cost reflected in a throughput improvement?	155
Why is there no improvement to transfer rate despite the transport cost being reduced?	156
Starting and stopping MQ channels protected by AT-TLS	157
AT-TLS start channel performance	158
AT-TLS stop channel performance	159
Should I use AT-TLS to provide encryption of my MQ channels?	159
Costs of Moving Messages To and From zOS Images	160
Non-persistent messages - NPMSPEED(FAST)	162
Persistent messages	163
4 System	165
Hardware	165
DASD	165
Maximum request/reply throughput (DS8900F)	165
Upper bound on persistent message capacity - DASD log data rate	166
What is the effect of dual versus single logging on throughput?	167
Will striped logs improve performance?	167
Should MQ for z/OS use log striping?	167
Will striped logs affect the time taken to restart after a failure?	167
Benefits of using zHPF with IBM MQ	168
When can it help with IBM MQ work?	168
Network	169

IBM MQ and zEnterprise Data Compression (zEDC)	171
Reducing storage occupancy with zEDC	171
Can I use zEDC with MQ data sets?	171
What benefits might I see?	171
What impact might I see?	172
How we set up for testing	172
What to watch out for	173
Measurements	173
IBM MQ and zEnterprise Data Compression (zEDC) with SMF	176
Data set encryption	177
Why use data set encryption	178
Data set encryption with the MQ queue manager	179
Active and Archive log encryption	180
Page set encryption	182
Shared message data set encryption	183
MQPUT to SMDS	184
MQGET - When the messages are read from SMDS buffers	185
MQGET - When the messages are read from local SMDS	185
MQGET - When the messages are read from remote SMDS	185
Comparing the cost of MQGETs from shared queue	186
Why are unencrypted gets more expensive than encrypted?	188
Summary of data set encryption costs with the MQ queue manager	189
zHyperWrite support for active logs	191
What is zHyperWrite?	192
Why does my log performance matter?	193
zHyperWrite test configuration	194
Reduced I/O time	195
Reduced elapsed time for MQ commit	196
Improved sustainable log rate	197
Impact to MQ queue manager costs	198
Impact of I/O limitations on dual active and dual archive logs on older hardware	200
Summary of zHyperWrite benefits	203
5 How It Works	204
Tuning buffer pools	204
Introduction to the buffer manager and data manager	204
The effect of message lifespan	205
Understanding buffer pool statistics	206
Definition of buffer pool statistics	208
Interpretation of MQ statistics	209
Observations on the problem interval	210
What was happening	211
Actions taken to fix the problem	211
Log manager	212
Description of log manager concepts and terms	212
Illustration of logging	213
When does a write to the log data set occur?	213
How data is written to the active log data sets	213
Single logging	213
Dual logging	213
Interpretation of key log manager statistics	214
Detailed example of when data is written to log data sets	214
MQPUT example	216
MQGET example	217

Interpretation of total time for requests	217
What is the maximum message rate for 100 000-byte messages?	217
6 Advice	219
Use of LLA to minimize program load caused throughput effects	219
Frequent use of MQCONN/MQDISC - for example WLM Stored Procedures	219
Frequent loading of message conversion tables	220
Frequent loading of exits - for example, channel start or restart after failure	220
Frequent loading of CSQQDEFV	220
System resources which can significantly affect IBM MQ performance	220
Large Units of Work	221
Application level performance considerations	222
Common MSGID or CORRELID with deep shared queues	223
Why is cost of MQGET higher when more than 5,200 messages have common identifier?	225
Frequent opening of un-defined queues	226
Frequent opening of shared queues	227
How can I tell if I am seeing first-open or last-close effects?	231
Can I reduce the impact from locking on my shared queues?	231
Is using an application to hold the queue open always appropriate?	232
Using GROUPID with shared queues	233
Comparing performance of GROUPID with CORRELID	233
Comparing performance of GMO options	234
Avoiding Get-Next when specifying GroupID	235
Using Message Selectors	236
Who pays for the cost of message selection?	236
Is there a good message selector to use?	236
How do I know if I am using a good message selector?	236
Message selector performance	237
Message selector performance with private queues	238
Message selector performance with shared queues	240
Checklist: Using client-based selectors	242
Temporary Dynamic (TEMPDYN) Queues	243
TEMPDYN queues - MQOPEN	243
TEMPDYN queues - MQCLOSE	243
7 Queue Information	245
Tuning queues	245
Queue option ACCTQ	245
Queue option DEFPRESP	245
Queue option DEFREADA	245
Queue option MONQ	245
Queue option PROPCTL	246
Maximum throughput using non-persistent messages	247
What factors affect non persistent throughput	247
Private queue	248
What is the maximum message rate through a single private queue ?	248
Throughput for request/reply pairs of private queues	249
Shared queue	251
Maximum persistent message throughput - private queue examples	252
Strict ordering - single reply application	252
Increasing number of reply applications	253
Maximum persistent message throughput - shared queue examples	254
Shared queue persistent message - CPU costs	255
Shared queue persistent message - CF usage	256

Message ordering - logical groups	258
Does size of group matter?	258
Large groups of small messages OR small groups of large messages?	258
Application tuning	261
How much extra does each waiting MQGET cost?	261
How much extra does code page conversion cost on an MQGET?	261
Event messages	261
Triggering	261
What is the cost of creating a trigger or event message?	261
8 Two / Three Tier configurations	262
Why choose one configuration over the other?	263
Cost on the z/OS platform	263
Achievable Rate	264
Number of connecting tasks	265
Measurements	265
9 IMS Bridge: Achieving best throughput	268
Initial configuration	269
How does the IMS bridge work?	269
Putting messages from IBM MQ into IMS	269
IMS putting reply messages to IBM MQ	269
Tuning the IMS subsystem	270
Use of commit mode	272
Commit Mode 0 (Commit-Then-Send)	272
Commit Mode 1 (Send-Then-Commit)	272
Understanding limitations of the IMS bridge	274
When do I need more message processing regions?	276
Understanding the trace reports - run profile	277
Understanding the trace reports – call summary	278
Understanding the trace reports – region summary report	279
IMS Control Region issuing checkpoints whilst monitoring running	280
Understanding the Trace reports – Region IWAIT Report	281
Understanding the trace reports – Program Summary Report	282
Understanding the trace reports – Program I/O Report	283
When do I need more TPIPEs?	286
10 Hardware Considerations	290
Example: LSPR compared to actual results	291
Overview of Environment: Workload	291
Batch Applications	292
Hardware	292
LSPR tables	292
Non-persistent in-syncpoint messages	293
11 MQ Performance Blogs	295

Chapter 1

Queue Manager

When installing IBM MQ for z/OS, it is important to consider the following configuration options and decide on the most appropriate definitions for your particular queue manager environment. You should consider these options before customizing the queue manager because it might be difficult to change them once the queue manager has been defined.

The following configuration options should be considered:

- Using appropriate queue manager attributes
- Log data set definitions
- Page set definitions
- Buffer pool definitions
- If you are using shared queues
 - Coupling Facility (CF) structure definitions
 - DB2 table definitions and associated buffer pool and group buffer pool definitions.
 - Shared message data size and usage.

This chapter describes the factors that should be taken into account when designing your queue manager environment.

Queue manager attributes

In a production environment for best performance, it is advisable that both global trace “TRACE(G)” and channel initiator “TRACE(CHINIT)” are disabled unless requested by level 3 Service. For further details of the impact on running with the queue managers global trace enabled, see “[Trace Costs](#)”.

Log data set definition

Before setting up the log data sets, review the following section in order to decide on the most appropriate configuration for your system.

Should your archive logs reside on tape or DASD?

When deciding whether to use tape or DASD for your archive logs, there are a number of factors that you should consider:

- Review your operating procedures before making decisions about tape or disk. For example, if you choose to archive to tape, operators must be available to mount the appropriate tapes when they are required.
- During recovery, archive logs on tape are available as soon as the tape is mounted. If DASD archives have been used, and the data sets migrated to tape using hierarchical storage manager (HSM), there will be a delay while HSM recalls each data set to disk. You can recall the data sets before the archive log is used. However, it is not always possible to predict the order in which they will be required.
- When using archive logs on DASD, if many logs are required (which might be the case when recovering a page set after restoring from a backup) you might require a significant quantity of DASD in order to hold all the archive logs.
- In a low usage system or test system, it might be more convenient to have archive logs on DASD in order to eliminate the need for tape mounts.

How to minimize recovery time

To minimize recovery time and avoid operational complexity it may be best to

- Keep as much recovery log as possible in the active logs on DASD, preferably at least enough for one day.
- Archive straight to tape.
- Page set image copy frequency should typically be at least daily.

Log shunting, introduced by WebSphere MQ version 6.0.0 (see WebSphere MQ for z/OS Concepts and Planning Guide GC34-6582), makes it unlikely that archive logs will be required after a queue manager failure as shunted log records contain sufficient information for transactional recovery.

However, media recovery from a page set or CF application structure failure still requires the queue manager to read all log records since the date and time of the last image copy of that pageset or CF structure backup.

There is some small CPU saving when reading from active versus archive log on disk, but the major objective is to take maximum advantage of available disk space.

The tuning variables are image copy frequency, dualling all image copies to avoid fallback to previous image copy and how much disk space can be made available for the active logs.

Should your installation use single or dual logging?

There is little performance difference between single and dual logging to write-cached DASD unless the total I/O load on your DASD subsystem becomes excessive.

If your DASD type is a physical 3390 or similar, you are advised to use dual logging in order to ensure that you have an alternative backup source in the event of losing a data set, including loss by operator error. You should also use dual BSDSs and dual archiving to ensure adequate provision for data recovery.

If you use devices with in-built data redundancy (for example, Redundant Array of Independent Disks (RAID) devices) you might consider using single active logging. If you use persistent messages, single logging can increase maximum capacity by 5 - 50% and can also improve response times.

If you use dual archive logs on tape, it is typical for one copy to be held locally, and the other copy to be held off-site for use in disaster recovery.

Other reasons for dual logging:

- Software duplexing gives separate names to datasets, which reduces the risk of manual error destroying data, e.g. deleting.
- Different names may map to different storage / management classes so one copy may be local and the other remote.

How many active log data sets do you need?

You should have sufficient active logs to ensure that your system is not impacted in the event of an archive being delayed.

In practice, you should have at least four active log data sets but many customers have enough active logs to be able to keep an entire day's worth of log data in active logs. For example, if the time taken to fill a log is likely to approach the time taken to archive a log during peak load, you should define more logs. You are also recommended to define more logs to offset possible delays in log archiving. If you use archive logs on tape, allow for the time required to mount the tape.

How large can your active logs be?

Prior to MQ version 8.0, when archiving to DASD, the largest single archive dataset supported was 65,535 tracks as IBM MQ does not support splitting an active log to multiple archive datasets on DASD and this is the size limit for data sets accessed with BDAM. This means that the maximum size of an active log is 65,535 tracks, approximately 3.5GB. This limit exists because the size of the log is restricted to the size of the largest archive log that IBM MQ for z/OS is able to access.

From version 8.0, BDAM is no longer used for archive access, so both active and archive logs on DASD may be up to 4GB in size.

If archiving to tape then the largest active log remains at 4GB regardless of MQ release.

Note: When the archive logs are written to DASD and have a primary allocation exceeding 65,535 tracks, (available from version 8.0.0) it may be necessary to ensure the archive data sets are allocated by a DFSMS data class that has a data set name type of LARGE or EXT. LARGE indicates that data sets in the data class are to be allocated in large physical sequential format. EXT indicates that data sets are to be allocated in extended physical sequential format. A setting of EXT is recommended, and is required for striping of data sets. If you specify EXT, also set the IFEXT (if extended) parameter to R (required) rather than P (preferred).

Note: We found that using archive logs of larger than 65,535 tracks resulted in additional CPU cost within the queue manager address space. It is suggested that archive logs larger than 65,535 tracks are only used when the queue manager is likely to be short of active logs. In the event of using these larger archive logs, some additional benefit may be gained by striping across multiple volumes.

How large should the active logs be?

Your logs should be large enough so that it takes at least 30 minutes to fill a single log during the expected peak persistent message load. If you are archiving to tape, you are advised to make the logs large enough to fill one tape cartridge, or a number of tape cartridges. (For example, a log size of 3GB cylinders on 3390 DASD will fit onto a 3592 tape with space to spare.) When archiving to tape, a copy of the BSDS is also written to the tape. When archiving to DASD, a separate data set is created for the BSDS copy. Do not use hardware compression on the tape drive as this can cause a significant impact when reading the tape backwards during recovery.

If the logs are small (for example, 10 cylinders) it is likely that they will fill up frequently, which could result in performance degradation. In addition, you might find that the large number of archive logs required is difficult to manage.

If the logs are very large, and you are archiving to DASD, you will need a corresponding amount of spare space reserved on DASD for SMS retrieval of migrated archive logs, which might cause space management problems. In addition, the time taken to restart might increase because one or more of the logs has to be read sequentially at start time.

Active log placement

High persistent message throughput typically requires that the active logs are placed on fast DASD with minimum contention from other data set usage. This used to mean there should be no other data set with significant use on the same pack as an active log. With modern RAID DASD the 3390 pack is logical with the physical data spread across multiple disk devices. However, the z/OS UCB (unit control block) for the logical pack may still be a bottleneck. UCB aliasing is available with the z/OS parallel access volumes (PAV) support enabled. You can then have several busy data sets on such a logical pack with good performance for each. This can be exploited to ease any active log placement problems. For instance, you could have the current active log on the same logical pack as the preceding active log. This used to be inappropriate as the preceding log would be read for archive offload purposes while the current active log is being filled. This would have caused contention on a single UCB even to a logical pack.

Where UCB aliases are not available, then ideally, each of the active logs should be allocated on separate, otherwise low-usage DASD volumes. As a minimum, no two adjacent logs should be on the same volume.

When an active log fills, the next log in the ring is used and the previous log data set is copied to the archive data set. If these two active data sets are on the same volume, contention will result, because one data set is read while the other is written. For example, if you have three active logs and use dual logging, you will need six DASD volumes because each log is adjacent to both of the two other logs. Alternatively, if you have four active logs and you want to minimize DASD volume usage, by allocating logs 1 and 3 on one volume and logs 2 and 4 on another, you will require four DASD volumes only.

In addition, you should ensure that primary and secondary logs are on separate physical units. If you use 3390 DASD, be aware that each head disk assembly contains two or more logical volumes. The physical layout of other DASD subsystems should also be taken into account. You should also ensure that no single failure will make both primary and secondary logs inaccessible.

Striped logs

The active logs can be striped using DFSMS. Striping is a technique to improve the performance of data sets that are processed sequentially. Striping is achieved by splitting the data set into segments or stripes and spreading those stripes across multiple volumes. This allows multiple UCBs and this in conjunction with HyperPAV can improve the logging rate achieved with messages larger than 4KB. Messages smaller than 4KB will only write a single 4KB page and will not exploit striping.

Striped archive logs

Prior to IBM MQ version 8.0.0, archive logs used BDAM and as a result could not be striped.

From IBM MQ version 8.0.0, archive logs stored on DASD may be striped. Striping the archive logs may result in an improvement in offload time.

Note: Remember that when moving from non-striped to striped archive logs, it is advisable to divide the PRIQTY and SECQTY values in the CSQ6ARVP macro by the number of stripes otherwise each stripe will be allocated with the specified size e.g.

CSQ6ARVP parameter	No stripes	4 stripes
-------------------------------	-------------------	------------------

UNIT	3390	3390
ALCUNIT	CYL	CYL
PRIQTY	5600	1400
SECQTY	100	25

Log data set pre-format

Whenever a new log data set is created it must be formatted to ensure integrity of recovery. This is done automatically by the queue manager, which uses formatting writes on first use of a log data set. This takes significantly longer than the usual writes. To avoid any consequent performance loss during first queue manager use of a log data set, use the log data set formatting utility to pre-format all logs. See the supplied sample job SCSQPROC(CSQ4LFMT).

Up to 50% of maximum data rate is lost on first use of a log data set not pre-formatted on our DASD subsystem. An increase in response time of about 33% with loss of about 25% in throughput through a single threaded application was also observed.

New logs are often used when a system is moved on to the production system or on to a system where performance testing is to be done. Clearly, it is desirable that best log data set performance is available from the start.

Log data set - RAID5 or RAID10

When running tests designed to put persistent messages to the log datasets we have found no discernible difference in performance when using DASD configured as RAID5 over similar tests using DASD configured as RAID10.

8-byte log RBA

From version 8.0, IBM MQ for z/OS improves the availability of the queue manager by increasing the period of time before the log needs to be reset by expanding the size of the log RBA (Relative Byte Address) from 6 to 8 bytes. This increase is such that over 64,000 times as much data can now be written before the log RBA needs to be reset.

In order to support the increased log RBA, additional data is logged. For example a typical request-reply workload using a 1KB request message and a 4KB reply message would need an additional 7% log space over a queue manager using a 6-byte RBA. Were the reply message to be 64KB, the addition space would be 2% compared to a queue manager using a 6-byte log RBA.

From version 9.3, IBM MQ for z/OS, new queue managers have by default 8-byte log RBA and are capable of having up to 310 active logs.

How much log space does my message use?

The following tables are provided to give guidance as to the size of logged data for a range of persistent messages. The logged data sizes are an average calculation based on using the process described below. Some additional logging may be seen to due internal queue manager tasks.

These measurements were taken when they were the only workload running against the queue manager. The logcopy dataset was large enough to contain all of the measurements performed. Some additional log writes may be performed when the logcopy file becomes full and the next logcopy dataset is used.

All messages used were persistent and the application specified whether the messages were put or gotten in or out of syncpoint.

How the data was gathered:

- Queue Manager is started.
- DISPLAY LOG issued, note the RBA.
- Perform known workload, e.g. put 10,000 messages to queue.
- DISPLAY LOG issued, again note the RBA.
- Determine how much log data written by change in RBAs.
- Determine logging size per transaction by dividing the delta of the RBA by the known number of transactions performed between DISPLAY LOG commands.

The measurements were run against local and shared queues, for a range of message sizes.

We measured putting and getting from the queues using 3 scenarios:

- 1 message is put or gotten out-of-syncpoint.
- 1 message is put or gotten followed by a MQCMIT.
- 10 messages are put or gotten in-syncpoint following by a MQCMIT. Note, the data in the charts below show the measurements per message put or gotten, so using the data from the Local Queue – Put table, the total size of logged data for the “Put*10, Commit” measurement for a 1 byte message was 751 bytes * 10, i.e. 7510 bytes.

Local Queues

Message Size (bytes)	Log Size in bytes Per message PUT		
	Put out of Syncpoint	Put then Commit	Put*10, Commit
0	1057	1057	751
1	1060	1060	751
1024	2088	2088	1789
4096	5331	5331	5052

Message Size (bytes)	Log Size in bytes Per Message GOT		
	Get out of Syncpoint	Get then Commit	Get*10, Commit
0	481	481	186
1	481	481	186
1024	492	492	178
4096	527	527	192

Shared Queues

Message Size (bytes)	Log Size in bytes Per Message PUT		
	Put out of Syncpoint	Put then Commit	Put*10, Commit
0	683	1230	768
1	683	1230	768

1024	1687	2276	1802
4096	4780	5357	4884
Message Size (bytes)	Log Size in bytes Per Message GOT		
	Get out of Syncpoint	Get then Commit	Get*10, Commit
0	62	662	188
1	62	662	188
1024	69	650	177
4096	76	630	185

What is my logging rate?

Consider a requirement where a request/reply application needs to process 1000 transactions per second and the request message is 1KB and the response message is 5KB. The workload is run using local queues.

We can use the above charts to answer a question in the form of bytes per second, i.e.

Application	Action	Size (bytes)
Requester	Put 1KB message out-of-syncpoint	1024 (message) + 1064
Server	Get and commit 1KB	492
Server	Put and commit 5KB	5120 (message) + 1300
Requester	Get 5KB message out-of-syncpoint	550
Total	data logged during transaction	9550

In order to sustain 1000 transactions per seconds, the system needs to log at $1000 * 9550$ bytes per second, **9.11MB/second**.

How much log space do I need when backing up a CF structure?

When backing up a CF structure, the messages on the queues (in the structure) and in the DB2 BLOB tables affect how much is logged. The DISPLAY LOG command can be used to determine how much log space was used as described previously, however the CSQE121I message that is logged following successful completion of structure backup also provides the same information, converted into MB, e.g.:

CSQE121I @QMGR CSQELBK1 Backup of structure APPL1 completed at RBA=00004BFFF8C, size 24 MB

The following table gives guidance as to how much log data is written when the BACKUP CFSTRUCT command is used for a queue of fixed depth but different size messages.

Message size (bytes)	Queue depth	Backup size as reported by CSQE121I (MB)	Size of backup per message (bytes)	Overhead per message (bytes)
0	50,000	24	505	505
1	50,000	24	506	505
1024	50,000	73	1538	514
4096	50,000	219	4642	546

Using the above data as a guide we can predict that 200,000 messages of 4KB in the structure being backup up would use approximately $200,000 * 4642$ bytes, i.e. **855 MB**.

How can we estimate the required log data rate for a system?

The approximate amount of data written to a IBM MQ log for a persistent message that is MQPUT and committed then MQGET and committed is approximately:

Message size (bytes)	MQPUT + MQGET local or client channel	MQPUT + send on message channel with achieved batchsize		Receive on message channel + MQGET with achieved batchsize	
		=1	=50	=1	=50
500	1,700	4,300	1,950	2,750	1,450
1,000	2,300	4,850	2,450	3,300	2,000
5,000	6,550	9,150	6,750	7,600	6,250
10,000	11,750	14,300	11,900	12,800	11,500
30,000	32,800	35,400	33,000	33,900	32,550
100,000	105,800	108,400	106,000	106,900	105555

Log data rate formulae If required a more detailed estimate may be derived from the following:

User message length + length(all headers) + 1000 bytes
--

Thus, for a 1000 byte persistent message put to and got from a local queue approximately 2300 bytes of data will be written to the IBM MQ log. Using the maximum sustainable DASD data rates given in the section “[Upper bound on persistent messages capacity – DASD log data rate](#)”, for 1000 byte messages we estimate that up to 112 MB / 2,300 bytes = 51,061 persistent messages/second can be processed on our DS8900F RAID-5 DASD subsystem; we have achieved this throughput in one

measurement scenario with enough concurrent processes, though there was an increased response time. On other DASD subsystems you may get a different maximum.

For long messages the log data requirement is further increased by about 150 bytes per page occupied by the message and all its headers.

For example a 10,000 byte user message requires three 4KB pages.

$10,000 + \text{header length} + 1000 + (3*150) = 11,750$ bytes of data (approximately) will be required on the IBM MQ log for such a message on a local queue.

There is also the following log data requirement for each batch of messages sent or received on a channel (except for batches consisting entirely of non persistent messages on an NPMSPEED(FAST) channel):

- Messages in batch=1
 - Log requires 2.5KB per batch for the sender
 - Log requires 1.3KB per batch for the receiver
- Messages in batch=50
 - Log requires 3.7KB per batch for the sender
 - Log requires 1.3KB per batch for the receiver

If most of your MQPUTs are done at a completely different time to most of your MQGETs then you should be aware that most of the log data is associated with the MQPUT rather than the MQGET. As an example, you may receive messages over channels (MQPUTs) all day and only process those messages (MQGETs) in an overnight batch job.

For throughput estimating purposes assume:

1. For MQGET the log data requirement is about 500 bytes for messages up to a user length of 10 KB. This increases linearly to about 1 300 bytes for messages of user length 100 KB.
2. For MQPUT the actual message, including header data, is placed on the log. To estimate MQPUT requirement calculate
Total log requirement (as above) - MQGET log requirement

NOTE: The above calculations only give throughput estimates. Log activity from other IBM MQ processes can affect actual throughput.

Page sets

When deciding on the most appropriate settings for page set definitions, there are a number of factors that should be considered. These are discussed in the following sections.

Page set usage

In the case of short-lived messages, few pages are normally used on the page set and there is little or no I/O to the data sets except at start time, during a checkpoint, or at shutdown.

In the case of long-lived messages, those buffer pool pages containing messages are normally written out to disk. This is performed by the queue manager in order to reduce restart time.

You should separate short-lived messages from long-lived messages by placing them on different page sets and in different buffer pools.

Size of page sets for a given number of messages

The maximum size of a pageset is:

- 64GB for V6.0.0 and subsequent releases
V6.0 performance for messages on such page sets is the same as that for existing 4GB page sets. Utility performance is also unchanged per GB processed.
- 4GB for prior releases

The number of messages fitting into a page set is approximately as shown in the following charts when using a version 6.0 or version 7.0.1 and subsequent release queue managers, assuming all messages are the same size.

Pages per message	Message size (user data plus all headers except MQMD)			Approx msgs per 4GB pageset	Approx msgs per 64GB pageset
	V6	V7.0.1 onwards as shipped	V7.0.1 onwards maxshortmsgso		
8	27992 - 32040	27924 - 31971	27924 - 31971	125K	2M
7	23942 - 27991	23876 - 27923	23876 - 27923	142K	2285K
6	19894 - 23941	19828 - 23875	19828 - 23875	166K	2666K
5	15845 - 19893	15780 - 19827	15780 - 19827	200K	3200K
4	11796 - 15844	11732 - 15779	11732 - 15779	250K	4M
3	7747 - 11795	7684 - 11731	7684 - 11731	333K	5333K
2	3698 - 7746	3636 - 7683	3636 - 7683	500K	8M
Messages per page					
1	1656 - 3697	0 - 3635	1568 - 3635	1M	16M
2	981 - 1655	N/A	892 - 1567	2M	32M
3	643 - 980	N/A	554 - 891	3M	48M
4	440 - 642	N/A	351 - 553	4M	64M
5	305 - 439	N/A	216 - 350	5M	80M
6	208 - 304	N/A	119 - 215	6M	96M
7	136 - 207	N/A	47 - 118	7M	112M
8	79 - 135	N/A	0 - 46	8M	128M
9	34 - 78	N/A	N/A	9M	144M
10	0 - 33	N/A	N/A	10M	160M

Version 7.0.1 changed the manner in which short messages are stored. Refer to “[Why WebSphere MQ for z/OS changed the way it stored small messages in V7.0.1.](#)” for the reasons for this change.

To revert to version 7.0.0 and earlier behaviour you can specify the following, either in the CSQINP2 DD or as a command against the queue manager:

REC QMGR TUNE(MAXSHORTMSGS 0)

You should allow enough space in your page sets for the expected peak message capacity. You should also specify a secondary extent to allow for any unexpected peak capacity, such as when a build up of messages develops because a queue server program is not running.

NOTE: Prior to V6.0.0 the application that causes page set expansion will have to wait until the expansion has completed. This can be many seconds depending on the secondary extent size.

With V6.0.0 a page set that is allowed to expand will begin expansion when the 90% full threshold is passed. While expansion is in progress each MQPUT to it will be delayed by a few milliseconds. This means that is it less likely that an application needing to exploit a large page set will receive a 2192 (media full) return code.

Where DASD space allows, initial allocation of the largest possible page set remains the best option to minimize the possibility of an application stopping for media full reasons.

Number of page sets

Using several large page sets can make the role of the IBM MQ administrator easier because it means that you need fewer page sets, making the mapping of queues to page sets simpler.

Using multiple, smaller page sets has a number of advantages. For example, they take less time to back up and I/O can be carried out in parallel during backup and restart. However, consider that this adds a significant overhead to the role of the IBM MQ administrator, who will be required to map each queue to one of a much greater number of page sets.

The time to recover a page set depends on:

- The size of the page set because a large page set takes longer to restore.
- The time the queue manager takes to process the log records written since the backup was taken; this is determined by the backup frequency and the amount of persistent log data (to all queues on all page sets) required to be read and processed.

Recovering page sets

A key factor in recovery strategy concerns the period of time for which you can tolerate a queue manager outage. The total outage time might include the time taken to recover a page set from a backup, or to restart the queue manager after an abnormal termination. Factors affecting restart time include how frequently you back up your page sets, and how much data is written to the log between checkpoints.

In order to minimize the restart time after an abnormal termination, keep units of work short so that, at most, two active logs are used when the system restarts. For example, if you are designing an IBM MQ application, avoid placing an MQGET call that has a long wait interval between the first in-syncpoint MQI call and the commit point because this might result in a unit of work that has a long duration. Another common causes of long units of work is batch intervals of more than 5 minutes for the mover.

You can use the `DISPLAY CONN` command to display the RBA of units of work and to help resolve the old ones. For information about the `DISPLAY CONN` command, see the IBM MQ Script Command (MQSC) Reference manual.

How often should a page set be backed up?

Frequent page set backup is essential if a reasonably short recovery time is required. This applies even when a page set is very small or there is a small amount of activity on queues in that page set.

If you use persistent messages in a page set, the backup frequency should be in the order of hours rather than days. This is also the case for page set zero.

In order to calculate an approximate backup frequency, start by determining the target total recovery time. This will consist of:

- The time taken to react to the problem.
- The time taken to restore the page set backup copy. For example, we can restore approximately 8GB of 3390 data per minute from and to DS8900F DASD using DFDSS. Using REPRO, the rate was 3.5GB per minute - increasing to 5.2GB per minute when zHPF enabled..
- The time the queue manager requires to restart, including the additional time needed to recover the page set.

This depends most significantly on the amount of log data that must be read from active and archive logs since that page set was last backed up. All such log data must be read, in addition to that directly associated with the damaged page set. When using fuzzy backup, it might be

necessary to read up to three additional checkpoints, and this might result in the need to read one or more additional logs.

When deciding on how long to allow for the recovery of the page set, the factors you need to consider are:

- The rate at which data is written to the active logs during normal processing:
 - The amount of data required on the log for a persistent message is approximately 1.3 KB more than the user message length.
 - Approximately 2.5 KB of data is required on the log for each batch of non fast messages sent on a channel.
 - Approximately 1.4 KB of data is required on the log for each batch of non fast messages received on a channel.
 - Non-persistent messages require no log data. NPMSPEED(FAST) channels require no log data for batches consisting entirely of non-persistent messages.

The rate at which data is written to the log depends on how messages arrive in your system, in addition to the message rate. Non-fast messages received or sent over a channel result in more data logging than messages generated and retrieved locally.

- The rate at which data can be read from the archive and active logs.
 - When reading the logs, the achievable data rate depends on the devices used and the overall load on your particular DASD subsystem. For example, data rates of over 100 MB per second have been observed using active and archive logs on DS8900F DASD.
 - With most tape units, it is possible to achieve higher data rates for archived logs with a large block size.

Why WebSphere MQ for z/OS changed how it manages small messages in V7.0.1

In version 7.0.1 of WebSphere MQ for z/OS, a change to the way that MQ handles small messages was implemented.

Version 6.0.0 would attempt to fit as many messages as possible into each 4KB page, which gave an increased capacity - for example it was possible to fit 10 messages of 32 bytes onto a single page, which coupled with a 64GB page set limit, meant that a single page set could hold 160 million messages.

There was a downside to this - because more than 1 message could be stored on a single page, it was difficult to know exactly when all of the messages on any particular page had been retrieved and the page became available for new messages. To resolve this problem, a scavenger task would be initiated every 5 seconds to scan all of the pages to identify which pages could be re-used.

As clock speeds have increased, the achievable transaction rate has also increased and allowing a 5 second period between scavenger tasks meant more pages become empty but not marked as available. It also means that the scavenger task has to do more work scanning the chains to determine if a page can be marked as reusable.

From version 7.0.1, small messages are stored one per page. This mean that a 64GB pageset can never contain more than 16 million messages (i.e. 1/10th of the capacity for 32 byte messages).

The benefit of this approach is that once the message has been gotten, the data page can be deallocated immediately - there is no need to wait up to 5 seconds for the scavenger task.

In a high throughput environment, this means that there is less build up of unscavenged messages - so messages are more likely to be found in buffers rather than being in the pageset.

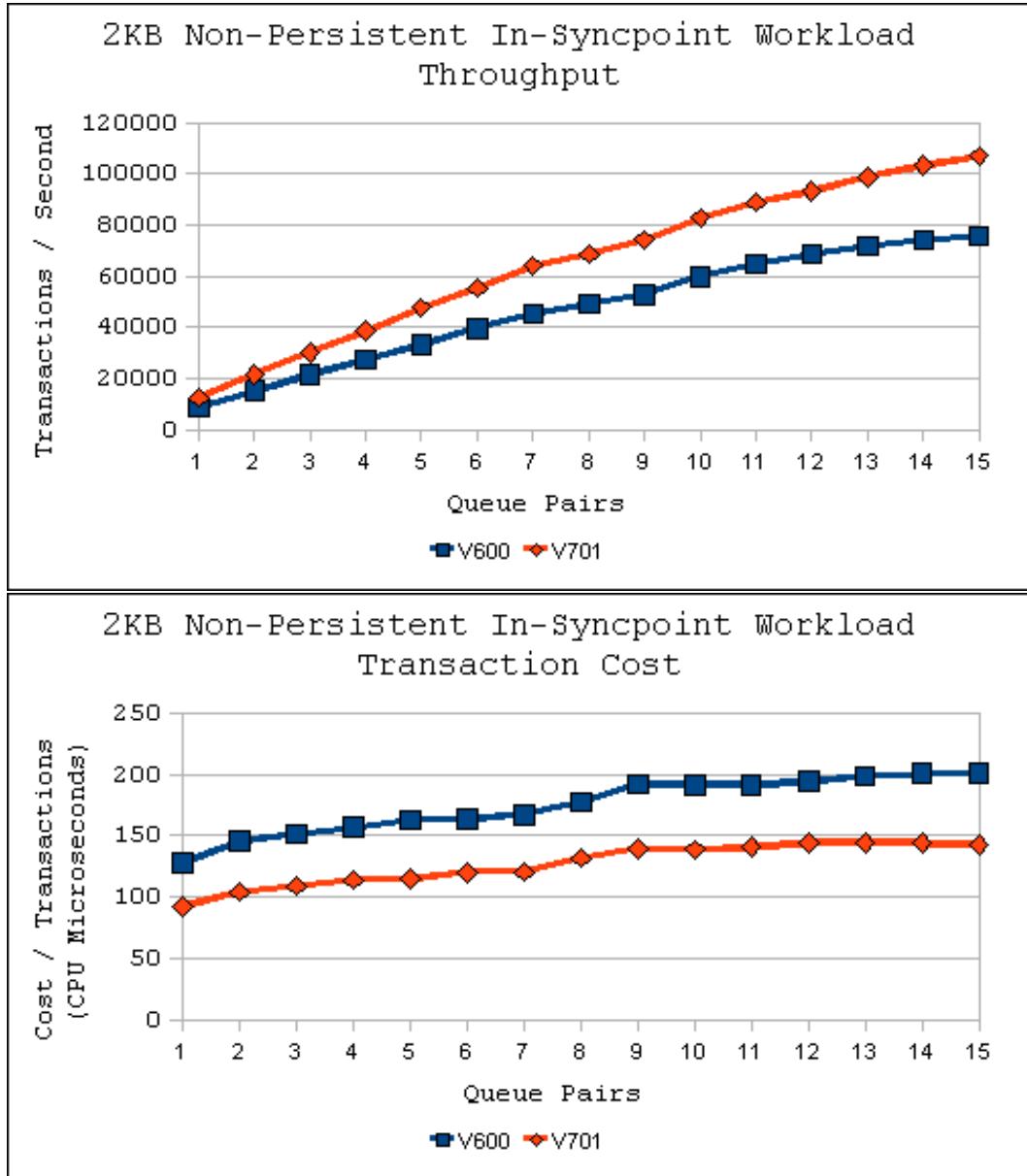
As an example of the benefit of the new approach, this is the output from a “DISPLAY USAGE” command against a V6.0.0 queue manager that has been running a non-persistent workload with 2KB messages where there is never more than 1 message on the queue. In this instance the test has been running for 17 seconds:

Page	Buffer	Total	Unused	Persistent	Nonpersistent	Expansion	
set	pool	pages	pages	data	pages	data	count
-	0	0	20157	19837	320	0	USER 0
-	1	1	268542	102704	29	165809	USER 0
-	2	2	214725	214725	0	0	USER 0

From this message, it can be seen that there are **165,809** pages that are marked as used - this is despite there being only 1 message on the queue at any time.

To further highlight the benefits, 2 charts are included. These show the results of a simple request/reply workload run using 2KB non-persistent private queue messages that are put and gotten in-syncpoint. The measurements start with 1 requester and 1 server performing a request/reply workload with a pair of queues defined on page set 1. As time goes on, more requesters and servers are added - using queues on separate page sets - so there is never more than 1 message on any particular page set.

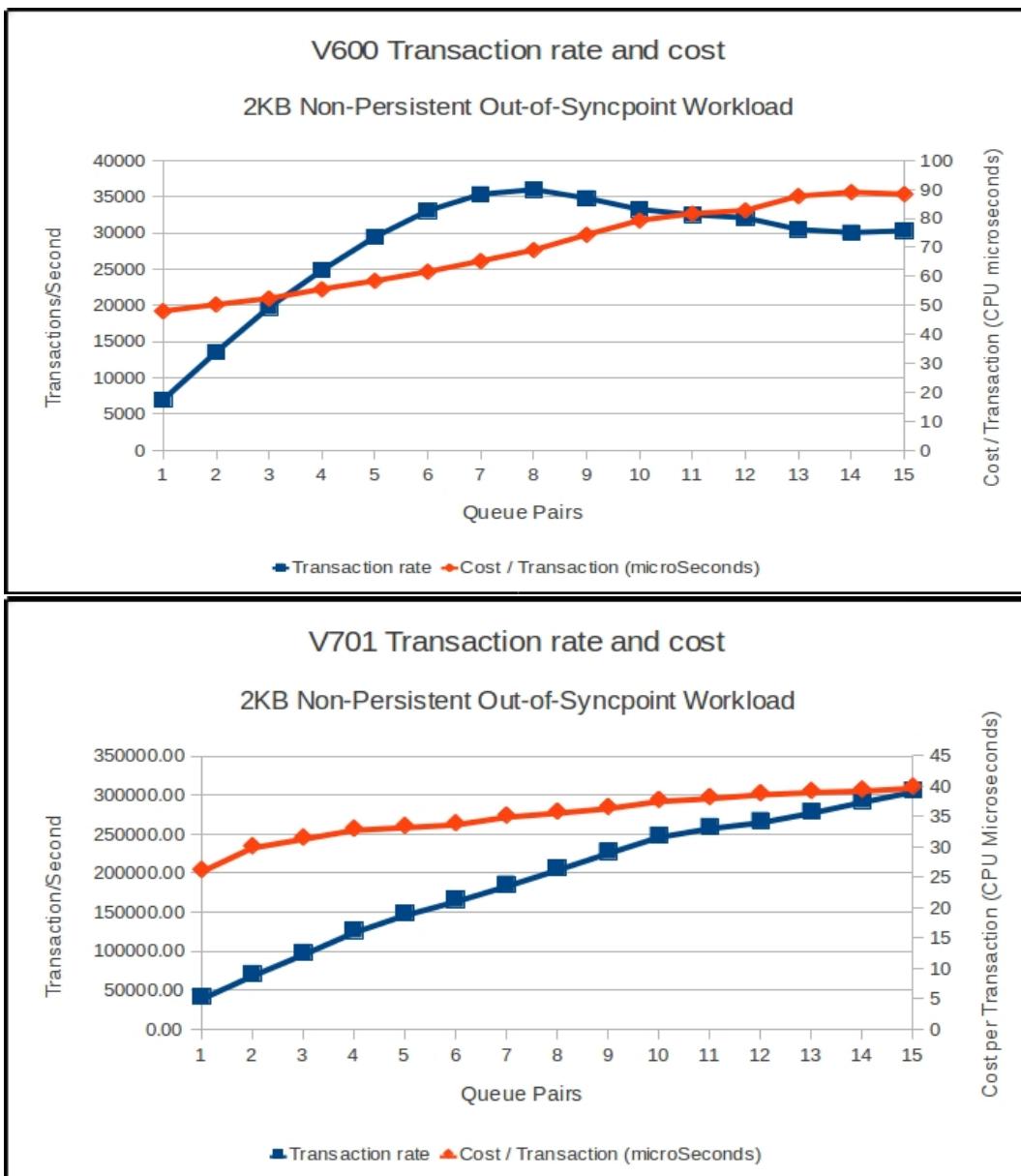
These tests are run on a single z/OS 1.12 LPAR of a zEnterprise 196 (2817) with 16 dedicated processors.



The comparison of non-persistent out-of-syncpoint workloads is even more marked but this is also assisted in the V7.0.1 measurement by the put-to-waiting getter being more likely to be successful. The following 2 charts have been included to compare the results of a simple request/reply workload run using 2KB non-persistent private queue messages that are put and gotten out-of-syncpoint.

The measurements start with 1 requester and 1 server performing a request/reply workload with a pair of queues defined on page set 1. As time goes on, more requesters and servers are added - using queues on separate page sets - so there is never more than 1 message on any particular page set.

These tests are run on a single z/OS 1.12 LPAR of a zEnterprise 196 (2817) with 16 dedicated processors.



NOTE: The y-axis scale is significantly different between the version 6.0.0 and version 7.0.1 charts.

The v6.0.0 measurement peaks at 37,500 transactions per second with a transaction cost between 50 and 80 microseconds. Despite 70% of the messages being put to waiting getter, the 15 pagesets all expanded multiple times - showing that the scavenger was unable to keep up.

The v7.0.1 measurement peaks at 300,000 transactions per second with a transaction cost between 25 and 40 microseconds. There were no pageset expansions nor were there any messages suggesting that the buffer pools were too small.

As the queue manager is not expanding the pagesets to store the unscavenged messages the maximum round-trip for these non-persistent out-of-syncpoint messages drops from 65 milliseconds to 1.1 milliseconds.

Summary

If capacity is the key requirement, then the v7.0.1 small message change may not be the most appropriate as the capacity for small messages can be as little as a tenth of the v6.0.0 capacity.

IBM MQ works more efficiently when queue depths are lower - messages in buffers cost less to get than messages in pagesets, getting messages from deep queues costs more than getting messages from shallow queues - and this small message change continues this theme with significant benefit to throughput.

Buffer pools

A buffer pool is an area of virtual storage in the private region of the queue manager address space. A BUFFPOOL definition gives the size in 4KB pages. Buffer pools are used to minimize I/O to and from page sets on disk. Thus both buffer pool sizes and actual usage can significantly affect the performance of queue manager operation, recovery, and restart. Each message queue is defined to a particular storage class (STGCLASS). Each STGCLASS is assigned to a page set (PSID). Each of the 100 page sets (0 to 99) is assigned to a particular buffer pool (BUFFPOOL). Thus any particular queue uses only one bufferpool and is ultimately stored in only one page set.

An outline of how buffer pools are managed is given in “[Introduction to the buffer manager and data manager](#)”.

Buffer pool default sizes

The following table shows suggested values for buffer pool definitions. Two sets of values are given; one set is suitable for a test system, the other for a production system or a system that will become a production system eventually.

Table: Suggested test and production definitions for buffer pool settings		
Definition setting	Test system	Production system
BUFFPOOL 0	1050 buffers (These were the supplied sample values until release V5.2. They are usually too small for production)	50 000 buffers These are the supplied sample values from release V5.2
BUFFPOOL 1	1050 buffers	20 000 buffers
BUFFPOOL 2	1050 buffers	50 000 buffers
BUFFPOOL 3	1050 buffers	20 000 buffers

Buffer pool usage

From V8.0.0 up to 100 buffer pools (0-99) may be defined, up from the previous limit of 16 buffer pools. This allows a 1:1 mapping of buffer pool to page set so that the buffer pools can be more finely tuned to their usage.

We recommend you use only 4 buffer pool definitions unless you:

1. Have class of service provision reasons for separating one set of queues from another
2. Have known queues which have different behaviour at different times or would otherwise be better isolated in their own individual buffer pools. This might be for their own performance benefit or to protect the performance of the other queues.

Prior to version 8.0.0, the storage for buffer pools was limited to the amount of space available within the overall queue manager address space virtual storage limit. From version 8.0.0, the buffer pools may be allocated using above the bar storage which provides multiple benefits including:

- Larger buffer pools can be allocated which reduce the need for writing to page set and being impacted by the increased I/O time to read/write from page set.
- Makes more space available in the queue manager's 31-bit storage for other things that haven't been moved above the bar e.g. more handles.

NOTE: The optimum value for these parameters is dependent on the characteristics of the individual system. The values given are only intended as a guideline and might not be appropriate for your

system. To make good use of the size recommendations you should consider separating buffer pool usage as follows:

1. A buffer pool for page set zero and the page set(s) containing system related messages.

Page set zero contains IBM MQ objects some of which must be frequently updated. For example, queue objects have to maintain a CURDEPTH value. Ideally, keep page set zero for these system-defined objects only. A crude estimate for the number of buffer pool pages required for the system objects in page set zero is half the number of objects.

The page set containing just system related messages, for example page set one, should also map to this buffer pool. System related messages are typically those in the SYSTEM.CLUSTER.* and SYSTEM.CHANNEL.SYNCQ queues

[It may be beneficial to put the SYSTEM.CLUSTER.* queues in their own buffer pool.](#)

Queues that can grow large unexpectedly (for example, the dead-letter queue) are particularly inappropriate for this buffer pool. We suggest you put such queues in the ‘everything else’ buffer pool.

This buffer pool should be large enough never to cross the less than 15% free threshold.

This will avoid unnecessary reads from the page set which will effect overall IBM MQ system performance if they are for system objects. A good starting point for the size of this buffer pool might be 50,000 pages.

Alter model queue definitions to point to a storage class other than SYSTEM so that they will not map to buffer pool zero.

2. A buffer pool for queues for your important long-lived messages.

When using buffers from 31-bit storage (below the 2GB bar), a good starting point for the size of this buffer pool might be 20,000 pages.

Long-lived messages are those that remain in the system for longer than two checkpoints, at which time they are written out to the page set.

While it is desirable within limits to define such a buffer pool so that it is sufficiently large to hold all of these messages, it is not advised to exceed 50,000 pages prior to version 8, otherwise there may be a concentration of the necessary page set I/O at checkpoints which might adversely affect response times throughout the system.

Version 8.0 is capable of running with larger buffer pools but be aware of the impact at checkpoint by reviewing the “[Deferred Write Processor](#)” section.

3. A buffer pool for queues for your performance critical short lived messages.

A good starting point for the size of this buffer pool might be 50,000 pages.

This means that you have to have only short lived messages in queues on page sets that you define to this buffer pool. Normally, the number of pages in use will be quite small, however, this buffer pool should be made large to allow for any unexpected build up of messages, such as when a channel or server application stops running.

In all cases, this buffer pool should be large enough never to cross the less than 15% free threshold.

4. A buffer pool for everything else.

You might not be able to avoid this buffer pool crossing the less than 15% free threshold. This is the buffer pool that you can limit the size of if required to enable the other three to be large enough. Queues such as the dead-letter queue, SYSTEM.COMMAND.* queues and SYSTEM.ADMIN.* queues should be placed here. A good starting point for the size of this buffer pool might be 20,000 pages.

See “[Definition of Buffer Pool Statistics](#)” for information about statistics to help monitor buffer pool usage. In particular, ensure that the lowest % free space (QPSTCBLS divided by QPSTNBUF) is never less than 15% for as many of the buffer pool usage types shown above as possible. Also

ensure where possible that the buffer pools are large enough so that QPSTSOS, QPSTSTLA and QPSTDPMC remain at zero.

NOTE: It is good practise to monitor buffer pool usage over a period of time to determine whether particular buffer pools of the appropriate size. If a monitored buffer pool never exceeds 50% used, it may be beneficial to reduce the size which would allow the administrator to allocate this storage to a buffer pool that is too small.

Using buffers allocated in 64-bit storage

Buffer pools can be defined with the attribute LOCATION(ABOVE) so that the buffers are allocated in 64 bit storage. This means that these buffer pools can be made much larger so that all message access is from storage, which can enhance application performance by reducing disk I/O to and from page set.

Page fixed buffers

Even with large buffer pools, for some kinds of processing where queue depths build up where it may not be possible to keep all of the data in the buffer pool. In these cases data is written to the page set during MQPUT and read from page set during MQGET processing. Where the highest levels of performance are required for this type of high I/O intensity workload, the buffer pool can be defined with PAGECLAS(FIXED4KB) which ensures that the buffers are permanently fixed in real storage so the overhead of the page-fix before the I/O and the unfix after I/O is removed.

Why not page fix by default?

If PAGECLAS(FIXED4KB) is specified and there is insufficient real memory available in the LPAR, the queue manager may fail to start or may impact other address spaces.

The effect of buffer pool size on restart time

Restart time is not normally dependent on buffer pool size. However, if there are persistent messages on the log that were not written to a page set before a queue manager failure, these messages are restored from the log during restart and are written to the page set at the checkpoint that occurs following restart completion. This should have no greater impact than any other checkpoint, and might complete before much application activity resumes.

If you reduce the buffer pool size significantly before restarting a system after an abnormal termination, this can lead to a one-time increase in restart time. This happens if the buffer pool is not large enough to accommodate the messages on the log thus requiring additional page set I/O during restart.

For further information on buffer pools, statistics and data manager, see section “[Tuning buffer pools](#)”.

Deferred Write Process

The deferred write process, also known as DWP or DWT, is a set of queue manager tasks that causes data held in buffer pools to be written to page set independently of the putting application(s).

Typically when a buffer pool reaches 85% full, DWP will start to write the oldest data from the buffer pool into the associated page set, making more buffer pool pages available to applications. This will continue until the buffer pool usage drops below 75% full.

Should the rate of data being put onto the buffer pool exceed the rate that DWP can write to page set and the buffer pool usage reaches or exceeds 95% full, then synchronous writes to page set occur, which can increase the elapsed time that an application takes to put a message.

What changed in IBM MQ version 8.0.0?

There were 2 changes in version 8.0 that affected the performance of the DWP.

- Increase in number of buffer pools from 16 to 100.
- Moving buffer pools from 31-bit storage to 64-bit storage. By moving the buffer pools above the 2GB “bar”, the total amount of storage allocated to buffer pools can be much larger than 1GB.

Prior to version 8.0, DWP would write up to 4 pages of data in a single I/O request. With the potential increase in the number of buffer pool pages that may need to be written, this version of DWP could take a substantial time to write the data to page set, and this increased the likelihood that applications putting messages would be affected by synchronous I/Os.

From version 8.0, DWP attempts to write 16 pages of data per I/O request and may have up to 64 parallel I/Os in-flight. This means that DWP can process up to 1024 pages (4MB) of data at a single time per buffer pool. With a maximum of 64 parallel I/Os per buffer pool, the use of HyperPAVs is recommended to exploit these parallel I/Os.

Oversized buffer pools

Oversized buffer pools are where the size of the buffer pool is larger than the pageset.

When using 64-bit storage for MQ buffer pools and there is a 1:1 mapping of buffer pool to page set, there may be benefits to performance if the buffer pool is sized to 105% of the size of the page set.

By oversizing the buffer pool to such an extent, the effect from page stealing DWP can be minimised. This also has the effect of reducing the impact from synchronous writes to page set as the buffer pool reaches its capacity.

The benefits from this will only be observed if your buffer pool is constantly reaching capacity and synchronous writes are occurring to page set.

How many DWP tasks are there?

There is a deferred write process for each buffer pool. With version 8.0’s change to support 100 buffer pools, this can have an effect on the number of tasks running within the queue manager.

The DWP is also used during checkpoint processing to process data that has been held in the buffer pool through 2 checkpoints. Once data in buffer pools has been through 2 checkpoints, the DWP task will write the messages to the page set.

How much data could DWP write at checkpoint?

Prior to version 8.0, the total amount of space used by buffer pools was likely to be less than 1GB of the queue managers 31-bit storage.

Since DWP will run when the buffer pool usage reaches 85%, the maximum amount of data that could be written at checkpoint was approximately 85% of 1GB, or around 220,000 pages.

In version 8.0, buffer pools may be much larger although there is no practical benefit in exceeding the size of the page set, which are limited to 64GB. In version 8.0 the maximum amount of data that can be written at checkpoint is actually limited by the log size, which has a maximum size of 4GB.

Note: Determining how much data is written by DWP at checkpoint is not as simple as the size of the log. As discussed in “[How much log space does my message use?](#)”, a 0-byte message put to a local queue will **log** 1057 bytes. However this is not necessarily the same as the amount of data that DWP will write to the page set. DWP will write full 4KB pages - regardless of the data size.

This means that a persistent message of up to 3635 bytes on the buffer pool will result in a 4KB write I/O to the page set.

For a 0 byte message, which caused 1057 bytes to be logged, there could be 3 messages logged per 4KB page, so a 4GB log could contain 12GB of buffer pool data that is eligible to be written by DWP at the second checkpoint.

Where there are multiple messages per commit, the amount of data logged is reduced. For example we have seen 10 messages of 0 bytes in a unit of work caused 751 bytes per message to be logged. In this instance, the log could write 5 messages per 4KB of log space. This has the potential to result in up to 20GB of buffer pool data that is eligible to be written by DWP at the second checkpoint.

What impact is there when DWP writes large amounts of data?

On our systems using DS8870 DASD, the DWP task was able to write data buffer pool to page set at a rate of between 300-600 MB/sec. When there is 4GB of data to be written to a single page set, this can mean that DWP is driving I/O for between 7 to 13 seconds. This increased load on the I/O subsystem can result in an impact on application workload until the DWP workload is complete.

In a system with less responsive DASD, the impact may be greater. For example if DWP is only able to write the 4GB of data at 100MB per second, there could be 41 seconds where an application workload using **persistent** messages is affected by the moving of data from buffer pool to page set.

The use of high performance FICON (zHPF) can reduce the impact on active workload when the queue manager DWP is initiated to write large amounts of data from buffer pool to page set.

DWP is also used at queue manager shutdown to write any persistent messages from buffer pool to page set but as there is less likely be MQ workload running on the queue manager, it is unlikely to be impacted.

Recovery

Achieving specific recovery targets

If you have specific recovery targets to achieve, for example, completion of the queue manager recovery and restart processing in addition to the normal startup time within xx seconds, you can use the following calculation to estimate your backup frequency (in hours):

Formula (A)

Backup frequency = (Required restart time * System recovery log read rate)
(hours) (in secs) (in MB/sec)

Application log write rate (in MB/hour)

For example, consider a system in which IBM MQ clients generate an overall load of 100 persistent messages per second. In this case, all messages are generated locally. If each message is of user length 1 KB, the amount of data logged per hour is of the order:

$100 * (1 + 1.3 \text{ KB}) * 3600 \text{ seconds} = \text{approximately } 800 \text{ MB}$

100 = message rate per second

$(1 + 1.3 \text{ KB})$ = amount of data logged for each 1KB persistent message.

Consider an overall target recovery time of 75 minutes. If you have allowed 15 minutes to react to the problem and restore the page set backup copy, queue manager recovery and restart must then complete within 60 minutes applying formula (A). This necessitates a page set backup frequency of at least every:

$$3600 \text{ seconds} * 60 \text{ MB per second} / 800 \text{ MB per hour} = 270 \text{ hours}$$

In this example, the backup time means that there would be 210GB of data to restore.

It is preferable to recover the data from active logs - and with MQ allowing a maximum log size of 4GB and 310 active logs, there is capacity for 1240GB of data to be held in active logs.

For example if the active logs are 2GB each and there are 10 logs, there are only 20GB of data available on the active logs. Based on this the backup would need to occur every 25 hours

$$20 \text{ GB} / 800\text{MB per hour} = 25.6 \text{ hours}$$

This assumes that all required log data is on active logs using DS89000F DASD. (If you can only read from the log at 0.5MB per seconds the calculation would be every:

$$3600 \text{ seconds} * 0.5 \text{ MB per second} / 800 \text{ MB per hour} = 2.25 \text{ hours}$$

If your IBM MQ application day lasts approximately 12 hours, one backup every 2 days is appropriate. However, if the application day lasts 24 hours, one backup every day is more appropriate.

Another example might be a production system in which all the messages are for request-reply applications (that is, a persistent message is received on a receiver channel and a persistent reply message is generated and sent down a sender channel).

In this example, the achieved batch size is one, and so there is one batch for every message. If there are 50 request replies per second, the overall load is 100 persistent messages per second. If each message is 1 KB in length, the amount of data logged per hour is of the order:

$$50((2 * (1+1.3 \text{ KB})) + 1.4 \text{ KB} + 2.5 \text{ KB}) * 3600 = \text{approximately } 1500 \text{ MB}$$

where:

50	= message pair rate per second
(2 * (1 + 1.3 KB))	= amount of data logged for each message pair
1.4 KB	= overhead for each batch of messages received by each channel
2.5 KB	= overhead for each batch of messages sent by each channel

In order to achieve the queue manager recovery and restart within 30 minutes (1800 seconds) requires that page set backup is carried out at least every:

$$1800 \text{ seconds} * 60 \text{ MB per second} / 1500 \text{ MB per hour} = 72 \text{ hours}$$

This assumes that all required log data is on DS8900F where the IDCAMS REPRO achieves 60MB/second.

Periodic review of backup frequency

You are recommended to monitor your IBM MQ log usage in terms of MB per hour (log size in MB over hours to fill that log).

You should perform this check periodically and amend your page set backup frequency if necessary.

Restart

How long will my system take to restart after a failure?

This section describes the factors relating to recovery that affect the restart time of a queue manager:

- What happens when a transaction processes an IBM MQ request. This describes the stages that a transaction goes through.
- What happens during the recovery phase of restart.
- How long each recovery phase of restart takes.
- An example of a calculation of time required for the recovery phase of restart.

The length of time that a queue manager takes to restart depends on the amount of recovery that it has to do. This covers recovery of applications that were processing persistent messages (transactional recovery), and media recovery which involves both recovery of a page set after a failure and recovery of persistent messages in a buffer pool that had not been written to a page set.

There are four stages to the recovery phase of the log during restart

1. Preparing for recovery.
2. Determining the status of IBM MQ and connected tasks at the point of failure. This includes identifying the lowest page set recovery RBA.
3. Bringing the system up to date: this might involve media recovery and forward transaction recovery of in-doubt and in-commit transactions.
4. Backing out changes for those tasks that were in-flight and in-backout at the time the queue manager stopped.

Other factors that may affect restart times include:

- Rebuilding indexed queues - refer to performance report [**RETIRED ** MP1G "WebSphere MQ for z/OS V7.0.1 Performance Report" \(archived\)](#) section on "Indexed queues". ** NEED A SECTION IN MP16 **

- Rebuild subscriptions.
- Structure recovery.

To understand what happens in these log recovery stages you need to understand what happens to a transaction as it updates recoverable resources (such as persistent messages) and commits them. You also need to understand what happens at when a checkpoint occurs. The next section gives a simplified description of these things.

What happens as a transaction processes an IBM MQ request Consider a CICS transaction that gets two persistent IBM MQ messages, updates a DB2 table, and commits the work by issuing the EXEC CICS SYNCPOINT command. This will use two-phase commit because resources in both IBM MQ and DB2 are updated.

Transaction activity	What happens within the queue manager
Transaction starts.	Internal IM MQ state: Initial state
MQGET request issued.	<p>The message is locked.</p> <p>Because a unit of recovery did not exist for this transaction:</p> <ul style="list-style-type: none"> • A unit of recovery is created. • The state is changed to "in flight". • A "Start unit of recovery" (Start UR) is moved to the log buffers. • The LOG RBA of this record is saved as STARTRBA in the unit of recovery record. <p>The "message deleted" flag is set in the message, and this change is moved to the log buffers.</p> <p>The current queue depth is decremented and this change is also moved to the log buffers.</p> <p>Final internal IBM MQ state: In flight.</p>
MQGET the second message.	<p>The message is locked.</p> <p>The "message deleted" flag is set in the message, and this change is moved to the log buffers. The current queue depth is decremented and this change is also moved to the log buffers.</p> <p>Final internal IBM MQ state: In flight.</p>
DB2 table update made	
EXEC CICS SYNCPOINT issued.	
CICS issues the first part of the two phase commit (the prepare request) to IBM MQ.	<p>A "Begin Commit phase 1" is moved to the log buffers.</p> <p>The state is changed to "In commit 1".</p> <p>Resource managers prepare for the commit.</p> <p>The state is changed to "In doubt".</p> <p>An "End commit phase 1" record is moved to the log buffers and the RBA of this record is saved in the ENDRBA field of the unit of recovery record</p> <p>The log buffers up to and including this record are forced to disk.</p> <p>Returns "OK" to CICS.</p> <p>Final internal IBM MQ state: In doubt.</p>

CICS issues the prepare to DB2.	See Note.
Providing both IBM MQ and DB2 replied OK, CICS issues the second part of the two phase commit (the commit) to MQSeries.	<p>A "Phase 1 commit to Phase 2 commit" record is moved to the log buffers.</p> <p>The state is changed to "In commit 2", the transaction is now in "Must complete" state.</p> <p>The log buffers up to and including this record are forced to disk.</p> <p>The state is set to "End phase 2 commit".</p> <p>An "End phase 2 Commit" record is moved to the log buffers.</p> <p>Any locked resources are unlocked.</p> <p>The unit of recovery is deleted.</p> <p>The state is set to "Initial state". Returns to CICS.</p> <p>Final internal MQSeries state: Initial state</p>
Providing both MQSeries and DB2 replied OK, CICS issues the second part of the two phase commit (the commit) to DB2.	NOTE: The calls to DB2 describe what logically happens. In practice, CICS optimizes the call to the last resource manager by passing the prepare and commit request together. In this example, DB2 was the last resource manager, in other cases IBM MQ might be the last resource manager, and so the prepare and commit requests would be treated as one request.
Note: If any resource manager is unable to commit, the requests are backed out	

What happens during a checkpoint

During a checkpoint, information about the following items is moved to the log buffers and the buffers are forced to DASD.

- Incomplete units of recovery.
- Recovery RBAs of all page sets.
- Unit of work descriptors for peer recovery.
- IMS bridge checkpoint and all Set and Test Sequence Number (STSН) information.

When a checkpoint occurs it starts a process for writing old changed buffer pool pages to disk. These disk writes are not part of the checkpoint itself.

What happens during the recovery phase of restart

The following figure shows an example of the messages produced during the recovery phase of restart: The messages are described in the following text.

Figure: Example queue manager job log showing recovery messages

```
CSQJ099I @12A LOG RECORDING TO COMMENCE WITH
STARTRBA=0211B7845000
CSQR001I @12A RESTART INITIATED
CSQR003I @12A RESTART...PRIOR CHECKPOINT RBA=0211B7842C44
CSQR004I @12A RESTART...UR STATUS COUNTS
IN COMMIT=1, INDOUBT=0, INFLIGHT=1, IN BACKOUT=0
CSQR007I @12A STATUS TABLE
T CON-ID      THREAD-XREF      S     URID      DAY    TIME
-----
S IYCPVC01 1119E2ACC3D7F1F50000032C C 0211B7843536 138 16:59:32
S IYCPVC01 1119E2ACC3D7F1F50000036C F 0211B7F88502 138 17:05:51
CSQR005I @12A RESTART...COUNTS AFTER FORWARD RECOVERY IN COMMIT=0,
INDOUBT=0
CSQR006I @12A RESTART...COUNTS AFTER BACKWARD RECOVERY INFLIGHT=0, IN
BACKOUT=0
CSQR002I @12A RESTART COMPLETED
```

Phase 1, restart begins

The queue manager displays message CSQR001I to indicate that restart has started.

```
CSQR001I @12A RESTART INITIATED
```

The CSQJ099I message preceding the CSQR001I message contains the approximate RBA at the point of failure.

```
CSQJ099I @12A LOG RECORDING TO COMMENCE WITH STARTRBA=0211B791A000
```

Phase 2, determine the state of the system at point of failure

1. The last checkpoint record is located from the BSDS.
CSQR003I @12A RESTART...PRIOR CHECKPOINT RBA=0211B7842C44
2. The recovery RBAs of each page set are read from the checkpoint records.
3. Page 0 of every page set is read to obtain the last logged RBA for the page set. The lowest RBA of all the page sets is used to determine where the log should be read from for media recovery.
4. An in-memory table is built from information in the checkpoint records of the tasks that were active at the checkpoint.
5. The log is read in a forward direction from the checkpoint to the point of failure. The in-memory table is updated as tasks complete or new tasks start.
6. Message CSQR004I displays how many tasks were in each state at the point of failure.

```
CSQR004I @12A RESTART...UR STATUS COUNTS IN COMMIT=1, INDOUBT=0,
INFLIGHT=1, IN BACKOUT=0
```

Phase 3, forward recovery

1. A list of all of the log ranges required for forward recovery is built from the list of tasks that are in doubt and in commit. **Note:** Because the log ranges between the STARTRBAs and

ENDRBAs are known from the units of recovery, only the logs that contain these ranges are used. This means that some archives might not be used.

2. The range of log records for the page sets is from the earliest RBA in the page sets up to the point of failure. In normal operation the earliest RBA is within three checkpoints before the point of failure. If it has been necessary to use a backup version of a page set, the earliest RBA might be considerably earlier.
3. These ranges are combined, and a list is built of the required log ranges and the corresponding active or archive logs. The logs are read from the lowest RBA to the highest.
Note: The logs are searched sequentially from the beginning of the data set until the start of the required log RBA is found, and then read to the end RBA.
4. For each task that is in commit, log records between the start of the unit of recovery and the ENDRBA are processed, and the changes reapplied.
5. For each task that is in doubt, log records between the start of the unit of recovery and the ENDRBA are processed, and the changes reapplied. Locks are obtained on resources as required in the same way that they are obtained during normal operation.
6. The log is read from the lowest RBA for all of the page sets and the data is replayed to rebuild the buffer pools (and the page sets if you are recovering from a backup version) as they were at the point of failure. All log records from the earliest required RBA are read, even if they do not apply to the particular page set.
7. These forward recovery steps are combined in one forward scan of the log.
8. Once this forward recovery has completed, transactions in "must-commit" state are completed. All in-doubt units of recovery stay in doubt (with any resources still locked) until they are resolved, (for example when the CICS region reconnects).
9. Message CSQR005I displays how many tasks are in commit or in doubt after forward recovery.

```
CSQR005I @12A RESTART...COUNTS AFTER FORWARD RECOVERY IN COMMIT=0,  
INDOUBT=0
```

Phase 4, backward recovery.

1. The log records for in-flight or in-backout transactions are processed and any changes made by these transactions are undone.
2. Every log record is processed from the last written record, back to the earliest RBA of any transaction that was in flight or in backout. You can determine this RBA from the URID specified in message CSQR007I for records with a status of F or A.

CSQR007I @12A STATUS TABLE					
T	CON-ID	THREAD-XREF	S	URID	DAY TIME
-	-	-	-	-	-
S	IYCPVC01	1119E2ACC3D7F1F50000032C	C	0211B7843536	138 16:59:32
S	IYCPVC01	1119E2ACC3D7F1F50000036C	F	0211B7F88502	138 17:05:51

3. Message CSQR002I is issued at completion.

```
CSQR006I @12A RESTART...COUNTS AFTER BACKWARD RECOVERY INFLIGHT=0, IN  
BACKOUT=0  
CSQR002I @12A RESTART COMPLETED
```

How long will each phase of the recovery take?

Most of the time taken to recover is spent processing the active or archive logs. This has two components:

1. Making the data sets available to the queue manager (for example mounting a tape or recalling a data set from HSM for archive logs).

This time depends on your operational environment and can vary significantly from customer to customer.

2. Reading the active and archive logs.

This depends on the hardware used, for example DASD or tape, and the speed at which data can be transferred to the processor. (On DS8900F DASD we achieved between 1200 MB per second reading the active logs backwards and 800 MB per second reading the active logs forwards.)

The figures below estimate the time needed to read the logs for each stage of recovery. You should include your estimate of how long it will take to make the media available.

Phase 1, restart begins The recovery environment is established, so this is very short.

Phase 2, determine the state of the system at the point of failure The active log that was being used at the point of failure is read from the start to locate the last checkpoint (this might be at the start of the log). The log is then read from this checkpoint up to the point of failure.

The point of failure could be just before another checkpoint was about to be taken, in the worst case, the point of failure might be at the end of a log. You can estimate how much data is written between checkpoints by calculating the size of a log divided by the number of checkpoints in the time it takes to fill this log. (In our tests, with a log of 1000 cylinders of 3390 it took approximately 5 minutes to read to the end of this log.)

When reading the log data sets, the I/Os are performed such that multiple pages are read in parallel and will benefit from z/OS features such as HyperPav being available.

The size of the messages being processed can impact the CPU time taken in this phase of recovery, as a workload consisting of small messages may result in a higher usage of CPU time (SRB) than the equivalent amount of data being recovered from a large message workload, which in a CPU constrained system may increase the elapsed time for restart.

Phase 3, forward recovery This covers three activities:

- Recovering in-commit and in-doubt tasks.
- Applying changes to a page set if a backup copy has been used.
- Rebuilding the page sets to the point of failure.

The duration of forward recovery is primarily dependent upon the number of pages that need to be read from page set as the I/O must be performed serially one page at a time, as well whether all of the changed data can be help in buffer pools.

The CSQR030I message reports the range of data that may be processed as part of forward log recovery. From this range, the maximum number of pages that may need to be read from disk can be calculated by dividing the difference by 4096.

On our test systems connected to DS8900F DASD, each read of a 4KB page took approximately 140 microseconds, which equates to 7140 pages per second (27.9MB/second).

When z High Performance Ficon (zHPF) was enabled, the page I/O time was reduced and gave approximately a 50% improvement in read rate.

The number of pages that need to be read will depend on the workload and in our tests saw instances of between 10% and 100% of the pages specified by the CSQR030I message having to be read from disk.

Phase 3, Recovering in-commit and in-doubt tasks Most time is spent reading the logs between the STARTRBA and ENDRBA for in-doubt and in-commit tasks. The log is read in a forward direction until the start RBA is located, and then the records are read and processed from that point. If a unit of recovery spans more than one tape, the whole of the first tape has to be read. For in-doubt tasks, any locks are re-obtained. Backwards log recovery is dependent upon the number of long running tasks that need to be recovered, but it should be noted that the data is not read backwards, instead the starting record is read directly and then the logs are processed forwards.

Tasks that have been through 2 checkpoints and have been log shunted may require less data to be read for recovery.

In our measurements using simple long running tasks, such as those that put a message and wait prior to committing the data, backwards recovery was of the order of 10 seconds or less but with more tasks to recover, backwards recovery may take longer.

Phase 3, Applying changes to a page set if a backup copy has been used If a backup copy of a page set is used, the log needs to be read from the point when the backup was taken, and all records read forward from that point. If you did not record the log RBA when the backup was taken, you can use the date and time when the backup was taken and look in a printout of the BSDS to find the archive logs that will be needed.

To calculate the time needed to read the logs:

1. Calculate the difference between the log RBA at the start of backup and the RBA at the point of failure (the STARTRBA value in message CSQJ099I).
If the backup was taken when the queue manager was active (a fuzzy backup), the RBA in the page set might be up to three checkpoints before the RBA when the backup was taken. This might be up to three additional archive logs.
2. Divide the RBA range (in MB) by the data rate your DASD or TAPE can sustain to calculate the time required to process this amount of data.

The worst case is when there is a damaged page set that was not backed up and has to be redefined. This sets the page set RBA to 0, and so all logs from the very first are required for recovery. In the example above, the previous checkpoint is 0211B7842C44. This is about 2300 GB of data. If this can be read at 56 MB per second, this will take almost 12 hours, but if the reads occur at 1GB per second the time is reduced to approximately 38 minutes.

If the page set had been backed up when the queue manager was down at the RBA of 021000000000, the required range of RBAs is 0211B7842C44 - 021000000000 (about 7000 MB of data). If this can be read at 56 MB per second, this is about 2 minutes plus the time to read from the checkpoint to the point of failure. You also need to add the time taken to make any archive log available, and include the time to restore the page set from a backup copy.

It is significantly faster to use DFDSS dump and restore than to use IDCAMS REPRO. For example, for a dataset of 1600 cylinders DFDSS dump took 15 seconds, and IDCAMS REPRO took 26 seconds. In both cases the backup dataset was the same size as the original dataset.

Rebuilding the page sets to the point of failure To recover a page set, up to three checkpoints worth of data has to be read from log. This is typically two checkpoints worth, but if the system failed when processing a checkpoint, three checkpoints worth of data needs to be processed. By

having at least three active log data sets, you will ensure that these records are read from the active logs, and not archive logs.

Phase 4, undo any changes for tasks that were in flight or in backout (backward recovery) The log has to be read in a backward direction from the point of failure to the earliest URID for those tasks that were in flight or in backout (a status of F or A). Reading backwards is considerably slower than reading forwards, (by a factor of 5 on some tapes, and on our DS8800 was half the rate of reading forwards), and is even slower if data compaction is used. In the example above, if the system failed when the RBA was 211F0000000, there is about 900 MB of data to be read. If the rate of reading backwards is 56 MB per second this will take about 16 seconds.

Example of calculation of restart time This section gives a worked example for the time taken to restart a queue manager that had tasks in different states when it stopped.

The calculations are to give an indication of the time taken to recover, for example, will a queue manager take 10 minutes or 10 hours to restart, rather than an accurate value.

Configuration

Dual logging was used, with 3 active logs in a ring. Each log was 100 cylinders of 3390 (about 74 MB each). Small logs were used for illustration purposes.

When archiving to tape, a Virtual Tape System (3494-VTS B16) was used. This looks like a 3490E to z/OS. Physically, the data is written to DASD before being ultimately staged to a 3590 tape.

Request/reply CICS transactions were used. The transaction put a 1000-byte persistent message to a server queue, and a batch server retrieved the message and put the reply back to the originator.

The in-doubt, in-flight, and in-commit transactions were achieved as follows:

- CEDF was used to suspend the in-flight transaction after the transaction had written a message and before the commit. Many transactions were run.
- The in-doubt transaction was created by suspending the application before the "Phase 1 commit to Phase 2 commit" was moved to the log buffers. Many transactions were run.
- The in-commit transaction was suspended the same way as the in-flight transaction. Many transactions were then run and the in-commit transaction was allowed to commit. The queue manager was then cancelled. Because the "End phase 2 commit" had not been written to the log data set, the transaction becomes in commit at system restart. If any other transaction had run that caused the log buffers to be forced to disk, the in-commit transaction would have had its "End phase 2 commit" written to the log data set, and would be seen as having completed at system restart.

The following table shows the tapes that were used, and their RBA ranges:

Number	STARTRBA	ENDRBA
1	0	000000009FFF
2	00000000A000	000004659FFF
3	00000465A000	000008CA9FFF
4	000008CAA000	00000D2F9FFF
5	00000D2FA000	000011949FFF
6	00001194A000	000015F99FFF
7	000015F9A000	00001A5E9FFF
8	00001A5EA000	00001EC39FFF
9	00001EC3A000	000023289FFF
10	00002328A000	0000278D9FFF

The log data on tapes 9 and 10 is still available on the oldest two of the ring of active logs. Active logs are always used in preference to archive logs where possible.

Output from when the queue manager was restarted after it was cancelled Phases 1 and 2, - estimate of the time needed

The following figure shows an example of output from a queue manager at restart:

08.36.13 CSQJ099I @V21A LOG RECORDING TO COMMENCE WITH
STARTRBA=0000296A8000
08.36.13 CSQR001I @V21A RESTART INITIATED
CSQR003I @V21A RESTART...PRIOR CHECKPOINT RBA=0000278DF333
08.36.27 CSQR004I @V21A RESTART...UR STATUS COUNTS
IN COMMIT=1, INDOUBT=1, INFLIGHT=1, IN BACKOUT=0
CSQR007I @V21A STATUS TABLE
T CON-ID THREAD-XREF S URID TIME
- - -
S IYCPVC02 1869E2ACC3D7F2F50000046C F0000000A1F55 .. 08:14:35
S IYCPVC02 1869EE04C9D5C3D40016274C D0000046280C4 .. 08:18:56
S IYCPVC02 1869F206C3D7F2F50034023C C000008E340AO .. 08:24:11

This shows that the time between issuing message CSQR001I and message CSQR004I (phase 1) is 14 seconds.

- The RBA in the CSQJ099I message (0000296A8000) is just after the point of failure.
- The last checkpoint is at 0000278DF333.
- The number of bytes between these two values is approximately 31 MB.
- If the log can be read at 2.7 MB per second, this will take about 12 seconds.

Phase 3, forward recovery - estimate of the time needed

- Tape 2 was read forwards and took 21 seconds. This is for the in-doubt transaction.
- Tapes 4 through 8 were read forwards; each tape took about 25 seconds to read. Tapes 9 and 10 were not needed because the data was still in active logs. This is for the in-commit transaction.

08.47.05 CSQR005I @V21A RESTART...COUNTS AFTER FORWARD RECOVERY
IN COMMIT=0, INDOUBT=1
CSQR007I @V21A STATUS TABLE
T CON-ID THREAD-XREF S URID TIME
- - -
S IYCPVC02 1869EE04C9D5C3D40016274C D0000046280C4 .. 08:18:56

The time taken between issuing message CSQR005I and message CSQR004I was 10 minutes 38 seconds, of which 6 minutes 30 seconds was spent mounting the tapes. The tapes were being read for just over 4 minutes. There is one task in commit and one in doubt.

The in-commit and in-doubt tasks are processed during forward recovery and the in-flight task is processed during backward recovery. There is no way of knowing when the last RBA was written for the in-doubt or in-commit units of recovery. For normal, well-behaved, transactions the time between the start of the unit of recovery and the commit request is short, but it might take a long time for the in-doubt unit of recovery to be resolved.

Processing the in-doubt transaction The in-doubt transaction was created by suspending the application before the "Phase 1 commit to phase 2 commit" was written to the logs. This was the only transaction running at the time so the RBA range between the STARTRBA and the point where the transaction was suspended was small.

The log has to be read from the "Start UR" to the "End commit phase 1". The STARTRBA is on tape 2 and the log has to be read sequentially to locate the STARTRBA. Then the log is read and processed up to the ENDRBA.

The START UR of the in-doubt transaction is 0000046280C4 and the STARTRBA of tape 2 is 00000000A000. The number of bytes to be read to locate the STARTRBA is:

0000046280C4 - 00000000A000 = 74MB

The test system can achieve a rate of 2.7 MB per second which means that it takes 27 seconds to read 74MB. The time taken to read the records for the unit of recovery up to the ENDRBA is small in this example. (In the example above, Tape 2 was read for 27 seconds.)

Processing the in-commit transaction The in-commit transaction put a message and was suspended the same way as the in-flight transaction. Many other transactions then ran. This suspended transaction was then allowed to commit, in the sense that the "end phase 2 commit" was moved to the log buffers. Before the buffers were written to the log data set the queue manager was cancelled. Because the "End phase 2 commit" has not been moved to the log data set, it becomes in commit at system restart.

- The STARTRBA of the transaction is on tape 4, and the whole of the tape has to be read, from RBA 000008CAA000 forward. It is read from the start of the tape up to the STARTRBA, and then from the STARTRBA up to the commit records.
- You might know how your applications behave, and know if they have long units of recovery. In this example the ENDRBA is at the point of failure (0000296A8000).
- The amount of data to be read is 0000296A8000 - 000008CAA000. This is about 547 MB. On the test system, this could be read in 202seconds (at a rate of 2.7MB per second). In the above example, this was read in about 240 seconds.

Recovery of the buffer pools The RBA from the page sets is within three checkpoints of the point of failure. The checkpoints were occurring when the log switched, so up to three active logs have to be read. Each log is 74MB, and at 2.7 MB per second will take about 27 seconds per log. For three checkpoints this will be 82 seconds. This activity occurs in parallel to the recovery of the in-commit and in-doubt tasks.

Total time for this forward recovery The time taken for forward recovery is the greater of:

- 27 seconds for the in-doubt unit of recovery plus 202 seconds for the in-commit unit of recovery
- 82 seconds for the recovery of the buffer pools which is approaching 4 minutes, and close to the actual elapsed time.

Phase 4, backward recovery - estimate of time needed The active logs were read backwards, so the archive logs on tape 10 and 9 were not needed.

08.47.05 CSQR007I @V21A STATUS TABLE				
T	CON-ID	THREAD-XREF	S	URID
-	-	-	-	-
S	IYCPVC02	1869EE04C9D5C3D40016274C	D0000046280C4	.. 08:18:56
08.47.54	IKJ56221I	DATA SET MQMDATA.TAPE.V21A1.A0000008	NOT ALLOCATED...	

This shows that it took 54 - 05 = 49 seconds to process the log records already in memory and to read the active logs backwards.

- Tapes 2 through 8 were read backwards taking between 85 and 140 seconds per tape, for a total of 12 minutes 31 seconds.

09.04.50 CSQR006I @V21A RESTART...COUNTS AFTER BACKWARD RECOVERY INFLIGHT=0, IN BACKOUT=0 CSQR002I @V21A RESTART COMPLETED

The time taken between issuing message CSQR006I and message CSQR005I is 17 minutes 45 seconds, of which 4 minutes was spent mounting the tapes and 12-13 minutes reading the tapes.

There is one task in flight with an RBA of 0000000A1F55. The log has to be read backwards from the point of failure to this RBA.

The point of failure is at 0000296A8000, so the amount of data to be read is 0000296A8000 - 0000000A1F55. which is nearly 700 MB. If the rate for reading data backwards is about 0.5 MB per second this will take about 1400 seconds (nearly 24 minutes). ¹

Total restart time The time for recovery is the total of the time in the three phases, that is 11 seconds + 202 seconds + 24 minutes (nearly half an hour) plus the time to mount tapes (for example, 13 tapes at 1 minute each) giving a total time of nearly 45 minutes.

NOTE: These numbers are on older hardware and the logs are deliberately small to highlight the recovery process and time.

¹

Using the Virtual Tape System, where the data had not been destaged to 3590 tapes, the data could be read at about 2.6 MB per second. When the data had been moved to tape, the average rate was about 0.6 MB per second, this includes the time to locate and mount the tape as well as reading it.

Messages which show the page set status at startup

A message CSQI049I is produced to show the RBA needed by each page set during forward recovery. If the two RBA values are different this is due to a page set backup being used.

```
09.37.29 CSQI049I @# Page set 0 has media recovery  
RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA  
09.37.29 CSQI049I @# Page set 1 has media recovery  
RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA  
09.37.29 CSQI049I @# Page set 2 has media recovery  
RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA  
09.37.29 CSQI049I @# Page set 3 has media recovery  
RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA
```

Messages to show progress during forward and backward recovery There are 4 messages:

CSQR030I

This shows the maximum log range required for forward recovery.

Note: Not every log in this range might be needed.

CSQR031I

This message is produced approximately every 2 minutes, and shows the current log RBA being processed during forward recovery. From two of these messages, and the RBA range in message CSQR030I, you should be able to calculate the maximum time the forward recovery phase will take. You will also need to include the time taken to make the archive logs available. Active and archive logs might be on different media and thus be processed at different rates.

CSQR032I

This shows the maximum log range required for backward recovery. Every log in this range will be needed.

CSQR033I

This message is produced approximately every 2 minutes, and shows the current log RBA being processed during backward recovery. From two of these messages, and the RBA range in message CSQR032I, you should be able to calculate the maximum time the forward recovery phase will take. You also need to include the time taken to make the archive logs available.

```
09.37.30 CSQR030I @# Forward recovery log range  
from RBA=0000479AF9FA to RBA=0000479B33A0  
09.37.30 CSQR005I @# RESTART...COUNTS AFTER FORWARD RECOVERY  
IN COMMIT=0, INDOUBT=0  
09.37.30 CSQR032I @# Backward recovery log range  
from RBA=0000479B33A0 to RBA=000008561B58  
09.38.43 CSQR033I @# Reading log backwards, RBA=00003E022000  
09.40.43 CSQR033I @# Reading log backwards, RBA=00002E39A11D  
09.42.43 CSQR033I @# Reading log backwards, RBA=00001E686466  
09.44.43 CSQR033I @# Reading log backwards, RBA=00000EAB6000  
09.45.31 CSQR006I @# RESTART...COUNTS AFTER BACKWARD RECOVERY  
INFLIGHT=0, IN BACKOUT=0  
09.45.31 CSQR002I @# RESTART COMPLETED
```

Messages about page set recovery RBA produced at checkpoints Message CSQP021I is produced during a checkpoint. It identifies the RBA stored in page 0 of the page set, and the lowest RBA of any page in the buffer pool for that page set. These values are usually the same.

```
09.45.31  CSQP018I @# CSQPBCKW CHECKPOINT STARTED FOR ALL BUFFER POOLS
09.45.31  CSQP021I @# Page set 0 new media recovery
RBA=0000479B4000, checkpoint RBA=0000479B4000
09.45.31  CSQP019I @# CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER
POOL 3, 2 PAGES WRITTEN
09.45.31  CSQP021I @# Page set 1 new media recovery
RBA=0000479B4000, checkpoint RBA=0000479B4000
09.45.31  CSQP021I @# Page set 2 new media recovery
RBA=0000479B4850, checkpoint RBA=0000479B4850
09.45.31  CSQP021I @# Page set 3 new media recovery
RBA=0000479B50A0, checkpoint RBA=0000479B50A0
09.45.31  CSQP019I @# CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER
POOL 2, 19 PAGES WRITTEN
09.45.32  CSQP019I @# CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER
POOL 1, 21 PAGES WRITTEN
09.45.32  CSQP019I @# CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER
POOL 0, 87 PAGES WRITTEN
09.45.32  CSQY022I @# QUEUE MANAGER INITIALIZATION COMPLETE
09.45.32  CSQ9022I @# CSQYASCP 'START QMGR' NORMAL COMPLETION
```

What happens during the recovery phase of restart when in a QSG

The following figures show examples of the additional messages produced during recovery phase of restart when the queue manager is in a queue sharing group.

Reconnecting to the structures in the coupling facility

```
12.14.45 CSQE140I @VTS1 CSQEEENFR Started listening for ENF 35
events for structure CSQ_ADMIN
12.14.45 IXL014I IXLCNN REQUEST FOR STRUCTURE PRF5CSQ_ADMIN
WAS SUCCESSFUL.  JOBNM: VTS1MSTR ASID: 0C91
CONNECTOR NAME: CSQEPRF5VTS101 CFNAME: AACF01
ADDITIONAL STATUS INFORMATION:

CONNECTION HAS BEEN REESTABLISHED
12.14.45 CSQE141I @VTS1 CSQEEENFR Stopped listening for ENF 35
events for structure CSQ_ADMIN
12.14.45 CSQE005I @VTS1 CSQECONN Structure CSQ_ADMIN connected
as CSQEPRF5VTS101, version=CA5DC40EEF336E88 0001043C
12.14.45 CSQE021I @VTS1 CSQECONN Structure CSQ_ADMIN
connection as CSQEPRF5VTS101 warning, RC=00000004 reason=02010407
codes=00000000 00000000 00000000
```

The queue manager attempts to reconnect to the named structure when it is notified that the coupling facility is available. The queue manager then connects to the named structure but issues warning with reason 02010407 (IXLRSNCODESPECIALCONN). These messages may be repeated for the application structures too.

CFLEVEL(5)

```
12.14.45 CSQE252I @VTS1 CSQEDSS4 SMDS(VTS1)
CFSTRUCT(APPLICATION1) data set MQMDATA.VTS1.SMDS.APPL1 space map
will be rebuilt by scanning the structure
12.14.45 CSQE255I @VTS1 CSQEDSS4 SMDS(VTS1)
CFSTRUCT(APPLICATION1) data set MQMDATA.VTS1.SMDS.APPL1 space map has
been rebuilt, message count 10240
```

The queue manager initiates the rebuilding of the shared message data set space map by scanning the coupling facility structure. Upon completion the CSQE255I message is logged showing that there are 10240 messages held on the structure.

Peer level recovery

```
12.14.45 CSQE011I @VTS1 CSQESTE Recovery phase 1 started for structure
CSQSYSAPPL connection name CSQEPRF5VTS101
12.14.45 CSQE013I @VTS1 CSQERWI1 Recovery phase 1 completed for structure
CSQSYSAPPL connection name CSQEPRF5VTS101
12.14.45 CSQE012I @VTS1 CSQERWI2 Recovery phase 2 started for structure
CSQSYSAPPL connection name CSQEPRF5VTS101
12.14.45 CSQE014I @VTS1 CSQERWI2 Recovery phase 2 completed for structure
CSQSYSAPPL connection name CSQEPRF5VTS101
12.14.45 CSQE006I @VTS1 CSQECLOS Structure CSQSYSAPPL connection name
CSQEPRF5VTS101 disconnected
```

Phase 1 of the peer level recovery process involves recovering units of work that were in progress at time of failure.

Phase 2 involves recovering failed queue managers the in-flight messages for that queue manager.

Worked example of restart times

After an abnormal shutdown, extra time is needed to recover the system from the log data sets, to rebuild the system to the point of failure, and then to commit or roll back the work.

In the measurements below, CICS applications put messages to a queue and a batch server program processes the message and puts a reply on the specified reply-to queue. The CICS application then gets the reply and terminates.

A certain number of CICS transactions were run and then the queue manager was cancelled and restarted.

During restart the duration between the start of the queue manager and key restart messages being produced were recorded.

Number of CICS transactions	Time between startup and CSQR001I	Time between CSQR001I and CSQR002I	Time between CSQR002I and CSQY022I
0	7 seconds	1 second	0.2 seconds
10000	7 seconds	59 seconds	0.2 seconds

There is a linear relationship between the time between messages CSQR001I and CSQR002I and the number of CICS transactions that have run between the last checkpoint and the system being cancelled.

If there are ongoing units of work that existed before the latest checkpoint when the system ended, MQ will have to go back further in the log to the start of the units of work, and read the log from that point. This will extend the restart time. This could happen with channels that have a very long BATCHINT time specified, or on which the remote end of a channel has failed and messages are in doubt.

A checkpoint is taken at the following times:

- When an active log fills and switches to the next active log.
- When the number of writes to log buffers (Write Wait + Write Nowait + Write Force in the log manager statistics) exceeds the number specified for the LOGLOAD parameter of CSQ6SYSP. The number of writes to log buffers is reset to zero after a checkpoint.
- When an ARCHIVE LOG command is issued, because this forces a log switch.
- At shutdown.

1000 transactions were run, and the log statistics show that the number of "writes to log buffers" was about 31 000, or 31 "write to log buffers" per transaction. This means that, with a LOGLOAD value of 450 000, we could run $450\ 000 / 31 = 14\ 516$ transactions before a checkpoint occurs. If the system fails just before a checkpoint, the time between restart messages CSQR001I and CSQR002I would be about 85 seconds. (10000 transactions take 59 seconds, so 14516 would take 85 seconds.) This gives a total restart time of about $7 + 85 + 0.2 = 92$ seconds.

Note: Different message sizes might have different numbers of "write to log buffers" per transaction.

Effect of the number of objects defined on start up time

Restart time is affected by the number of objects that are defined because these are read in and validated during startup.

Number of objects defined	Time between startup and CSQR001I	Time between CSQR001I and CSQR002I	Time between CSQR002I and CSQY022I
140	7	1	0.2
4140	7	3.6	0.2
14484	7	7.6	0.2

With 14 484 objects defined, the default allocation of 1050 buffers for buffer pool 0 is too small. After the size of the buffer pool had been increased, the buffer pool statistics showed that 1230 buffers had been used.

Tuning

Performance implications of very large messages

The use of very large messages is likely to impact performance in the following ways:

- Page set I/O is more likely with very large messages than with the same amount of data in smaller messages. Buffer pools are much more likely to be so full that synchronous output to page sets is required. This significantly slows all applications using an affected buffer pool.
- For persistent messages the log I/O load will be great and other smaller messages are likely to be significantly slowed waiting for log I/O or even just space in log buffers. Ensure that log parameter OUTBUFF is set at its maximum (4000).
- Increased virtual storage usage in applications and in the IBM MQ channel initiator.
 - This is likely to cause increased real storage usage in applications and IBM MQ buffer pools.
 - The maximum number of channels all transmitting 100-MB messages is unlikely to exceed 15 because of virtual storage limitations. The use of BATCHSZ(1) is recommended for any channel transmitting very large messages

These considerations could cause an increase in elapsed time and CPU cost for every message in your queue manager compared to using the same amount of data in several smaller messages.

Queue Manager attribute LOGLOAD

What is LOGLOAD?

LOGLOAD is defined as the number of log records written by IBM MQ between the start of one checkpoint and the next.

Checkpoint processing will begin with the first of:

- the LOGLOAD threshold is reached.
- the end of the active log is reached.
- the queue manager is stopped.

What settings are valid for LOGLOAD?

Valid values for LOGLOAD range from 200 up to 16 million log records.

The value of LOGLOAD can be changed by either of the following methods:

- use the SET SYSTEM command. To permanently change this, put the SET SYSTEM command into a data set in the CSQINP2 concatenation.
- update the CSQ6SYSP parameter and regenerate the parameter module.

What is an appropriate value for LOGLOAD?

In our measurements the best performance, in terms of transaction cost and throughput, have been obtained with a LOGLOAD in the range of 4 million to 16 million.

When might a lower LOGLOAD be appropriate?

There may be occasions where a LOGLOAD of less than 4 million may be appropriate to your system, for example:

- When restart time is defined by an SLA. See “[Restart](#)” for more information.

- When using IBM InfoSphere Data Replication for DB2 for z/OS as part of an Active-Active solution and the system uses **multiple outbound IBM MQ channels** defined between the capture and apply queue managers.
 - In this instance we found that a lower LOGLOAD of 100,000 helped reduce instances where one transmission queue might achieve better throughput due to a more favourable log force frequency.

What happens at checkpoint

When a checkpoint is driven, the Deferred Write Process (DWP) will be invoked to write data that has been held in the buffer pool through 2 checkpoints from the buffer pool to the page set. More information can be found in the section titled “[Deferred Write Process](#)”.

Note: There can be a significant difference in the rate that data can be written by DWP depending on the IBM MQ release. We saw an individual DWP writing data to page set at the following rates:

- **V710:** 25MB per second
- **V800:** 300-600MB per second

This increase in write rate in IBM MQ version 8.0.0 is due in part to an increase in the number of pages written per I/O request (from 4 to 16) and also due to the number of parallel I/Os (up to 64), which can have an impact on the I/O subsystems response times.

Impact of LOGLOAD on workload

The value of LOGLOAD can have an impact on the throughput (messages per second) and the cost of those messages in the queue manager address space.

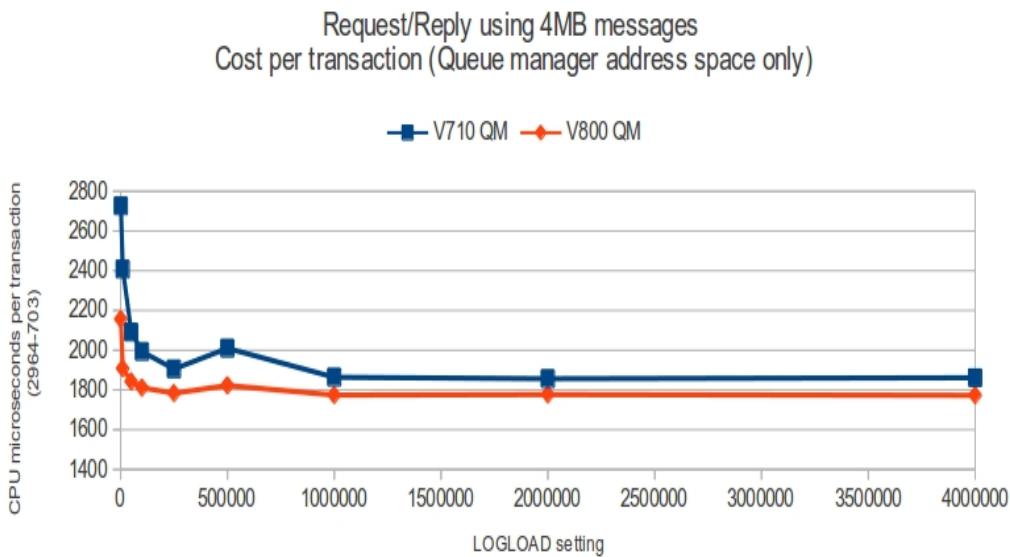
Measurements were run on V710 and V800 queue managers and used the following scenarios:

- **Low queue depth** - request/reply workload where the queue depths are low.
- **High queue depth** - deep queue processing where the queue was pre-loaded with messages and subsequently messages were put and got in random order whilst maintaining a high queue depth.
- **High variance in depth** - sending messages between queue managers using channels. E.g. where the rate that the messages arrive on the transmit queues exceeds the rate at which the messages can be got and sent across the network. This results in the transmit queues changing from empty to deep, until the point where the putting task completes, which then allows the transmit queue depth to drop to 0 as the channel initiator completes sending all of the messages.

Low queue depth

For both the V710 and V800 measurements there was little impact from LOGLOAD with messages up to 100KB.

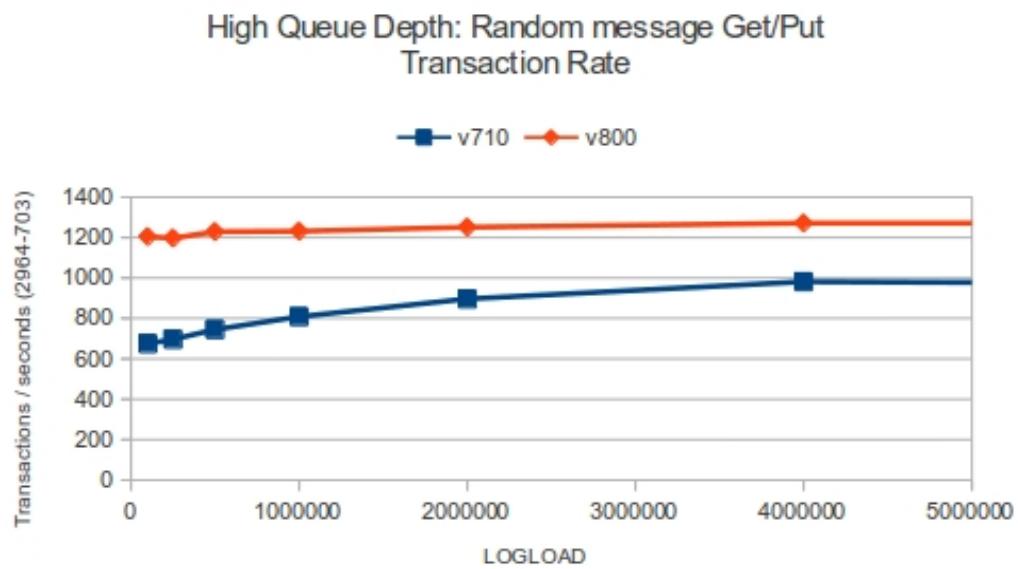
For larger messages, a small LOGLOAD value did impact the cost per transaction in the queue managers’ address space and as such, the optimal value for LOGLOAD would be 1 million upwards.



High queue depth

Optimum performance for this workload type was observed when the LOGLOAD was in the range 4 million to 16 million.

The following chart shows the achieved transaction rate with 100KB messages that are randomly got using a CORRELID and then put back to the queue.



In the V710 measurement, the largest impact is to the elapsed time of the MQPUT where it is delayed for log write.

Note: For this workload type, the V800 queue manager is using 64-bit buffer pools and benefits from the more intensive DWP and as a result is less affected by log writes as there are buffers available.

High variance in depth

Optimum performance for this workload type was observed when the LOGLOAD was in the range 4 million to 16 million but there were exceptions which include:

- *Sending data across multiple channels where the applications have a dependency of message ordering across the channels*, such as when using IBM InfoSphere Data Replication for DB2 for z/OS with **V710** queue managers which may benefit from a LOGLOAD at the lower end of the scale. In our measurements a LOGLOAD of 100,000 showed the most reliable performance largely due to reducing the effects of log-forces being skewed towards one channel, resulting in a more even balancing of log writes - which led to messages being sent at similar rates instead, and ultimately meant that messages became available for processing by the apply task sooner. In test measurements this had the effect of reducing the elapsed time for a fixed size workload by 12% when compared with a LOGLOAD in the range 4-16 million.

Impact of LOGLOAD on log shunting

Long running transactions can cause unit of work log records which span log data sets. IBM MQ handles this scenario by using log shunting, a technique which moves the log records to optimize the quantity of log data retained and queue manager restart time.

When a unit of work is considered to be long, a representation of each log record is written further down the log. This is known as “log shunting”.

At each checkpoint, whether triggered by reaching the end of an individual active log or by reaching the LOGLOAD threshold, long running transactions that are eligible for log shunting will be shunted. A smaller LOGLOAD value can impact the number of transactions being shunted and can also increase the amount of data being logged - which in turn can result in the checkpoints driven by a small LOGLOAD value to occur more frequently.

Impact of LOGLOAD on restart

The time taken to start a queue manager can be divided into four parts:

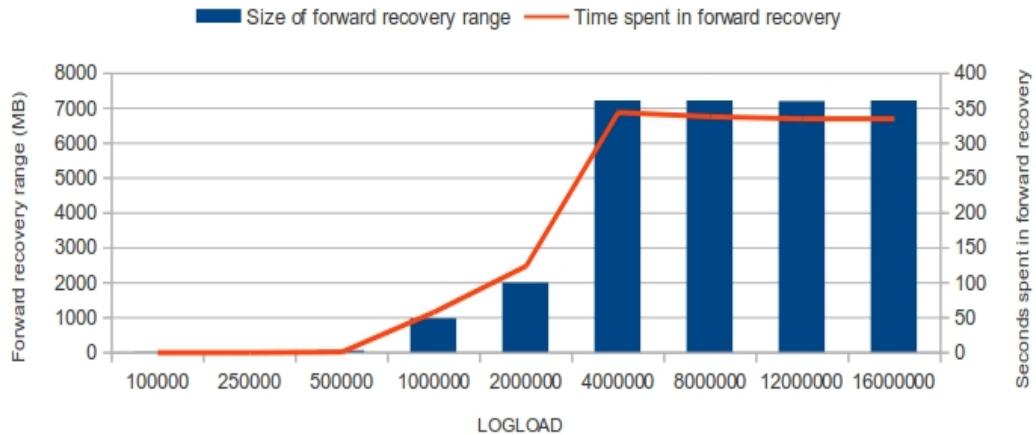
1. The time taken to load the MQ modules and for each component to initialize. Message CSQR001I is issued when this phase is complete.
2. The time taken to process the logs and recover any in-flight work; after a normal shutdown this work is very small. Message CSQR002I is issued when this phase is complete.
3. The time taken to read every object definition from page set 0 and to perform a consistency check on it. Message CSQY022I is issued when this phase is complete.
4. The time taken to process the statements in CSQINP2. Message CSQ9022I is issued when this phase is complete.

The setting of LOGLOAD can affect the amount of work that needs to be completed when processing the logs. The time spent in forwards recovery is largely influenced by the number of pages that need to be read from the page set(s). This is discussed in more detail in the “[How long will each phase of recovery take?](#)” section.

The following chart gives a guide to the recovery times as LOGLOAD increases, resulting in increased reads from page set.

QM Restart: Forward recovery time as more data needs recovering

Measurement on 2964-703 connected to DS8870 DASD where I/O time was 233 microseconds per page



Notes on chart: The proportion of pages read relative to the range of data specified by the CSQR03OI message varies. In this example the variation is between 10% (for LOGLOAD 500,000) and 100% (for LOGLOAD 1,000,000). In these extreme cases the amount of data specified by the CSQR03OI message was not particularly large, i.e. less than 1GB but when LOGLOAD was 4 million or higher, the range of data was 7GB of which 80% required reading, resulting in forward recovery times of 340 seconds.

Use of MQ Utilities

If possible avoid setting MAXUMSGS high. The number of MQ operations within the scope of a single MQCMIT should usually be limited to a reasonably small number. For example, you should not normally exceed 100 MQPUTTs within a single MQCMIT. As the number of operations within the scope of one MQCMIT increases the cost of the commit increases non-linearly because of the increasing costs involved in managing very large numbers of storage blocks required to enable a possible backout. So for queues with many tens of thousands of messages it could be very expensive to set MAXUMSGS greater than the number of messages and use CSQUTIL utility functions like COPY or EMPTY.

The 9.3 Administration Reference states, in the context of CSQUTIL utilities,

Syncpoints:

The queue management functions used when the queue manager is running operate within a syncpoint so that, if a function fails, its effects can be backed out. The queue manager attribute, MAXUMSGS, specifies the maximum number of messages that a task can get or put within a single unit of recovery.

The utility issues an MQCMIT call when the MAXUMSGS limit is reached and issues the warning message CSQU087I. If the utility later fails, the changes already committed are not backed out.

Do not just rerun the utility to correct the problem or you might get duplicate messages on your queues.

Instead, use the current depth of the queue to work out, from the utility output, which messages have not been backed out. Then determine the most appropriate course of action. For example, if the function is LOAD, you can empty the queue and start again, or you can choose to accept duplicate messages on the queues.

To avoid such difficulties if the function fails, there are two options:

- Temporarily increase the value of MAXUMSGS to be greater than the number of messages in:
 - The number of messages in the queue, if you are working with a single queue.
 - The longest queue in the page set, if you are working with an entire page set.
- Use the utility to LOAD the messages to a temporary queue. Then use the MQSC MOVE command to move the messages from the temporary queue to the target queue.
 - The LOAD and MOVE approach may take longer but moves the messages in a number of small units of work so is more efficient in terms of CPU cost.
 - Example move command: MOVE QL(tempq) TOQLOCAL(targetq) TYPE(ADD)

IBM MQ Utilities: CSQUTIL

The CSQUTIL utility program is provided with IBM MQ to help perform backup, restoration and re-organization tasks and to issue MQ commands.

These tasks include page set management functions including formatting and increasing the size of the page sets.

In order to increase the size of the page set, it is necessary to create the new page set and format it using the FORMAT function and then copy the contents of the existing page set into the new page set, using the COPYPAGE function.

The FORMAT and COPYPAGE functions may be performed within the same CSQUTIL job step or in separate steps.

FORMAT

The FORMAT function is used to format page sets, initialising them such that they can be used by the queue manager. On our system, each gigabyte of data took approximately:

- 0.35 CPU seconds
- 15 seconds elapsed

This means that to format a 4GB page set, it took 1.4 CPU seconds (on a 3-processor 8561-7G1 with DS8900F DASD) and 1 minute elapsed for the job to complete.

Enabling zHPF reduced the time for format a 4GB page set to 40 seconds but made no difference to the cost of the format.

The queue manager does have the capability to expand and format a page set whilst it is active, in the situation when it does not have sufficient capacity. In order to perform this expansion, the queue manager will slow any MQPUTs to queues on the page set whilst the expand and format takes place.

COPYPAGE

The COPYPAGE function is used only for expanding page sets to copy one or more page sets to a larger page set. The entire source page set is copied to the target page set, regardless of how many messages there are on the page set.

On our system, copying each gigabyte of page set took approximately:

- 0.91 CPU seconds
- 11.3 seconds elapsed

This means that to format a 4GB page set, it took 3.64 CPU seconds (on a 3-processor 8561-7G1 with DS8900F DASD) and 45 seconds elapsed for the job to complete.

RESETPAGE [FORCE]

The RESETPAGE function is like the COPYPAGE function except that it also resets the log information in the new page sets.

The “**RESETPAGE**” function:

- Does a copy and reset, using a source page set and a target page set.
- The target page set should have been opened previously, preferably using the FORMAT command, otherwise the step may fail with a "CSQU156E GET failed for CSQTxxxx data set. VSAM return code=00000008 reason code=00000074" message.

- The target page set size should be equal to or greater than the current size of the source page set including any expansions. The RESETPAGE function does not have the ability to expand the target page set and instead will fail before attempting the copy.

On our lightly loaded system (on a 3-processor 8561-7G1 with DS8900F DASD), each gigabyte of page set took approximately:

- 1.25 CPU seconds
- 23 seconds elapsed

This means that to RESETPAGE a 4GB pageset it took 5 CPU seconds and 93 seconds for the job to complete.

The “**RESETPAGE FORCE**” function:

- Does a reset in place, using only the source page set.
- This means that the page set will be of the appropriate size.
- As the same page set is being accessed for read and write operations, the rate of reset is significantly lower than the "RESETPAGE" option.

On our lightly loaded system (on a 3-processor 8561-7G1 with DS8900F DASD), each gigabyte of page set took approximately:

- 1.16 CPU seconds
- 37.5 seconds elapsed

This means that to RESETPAGE a 4GB pageset it took 4.64 CPU seconds and 2 minutes 30 seconds for the job to complete.

Enabling zHPF reduced the time to RESETPAGE FORCE on a 4GB page set to 1 minute 37 seconds but made no difference to the cost of the format.

Conclusions for RESETPAGE

- In order to ensure that queue manager down-time is minimised, it is advisable to use "RESET-PAGE" rather than "RESETPAGE FORCE", unless there is insufficient storage for a second set of page sets for the queue manager.
- If multiple page sets are to be reset, they should be processed as separate jobs run in parallel.
- Given that these jobs can be long running, it is advisable to check the service class is appropriate and the priority is not degraded over time.

Queue Manager Trace

For guidance on trace options for the channel initiator i.e. trace(chinit), accounting class 4 and statistics class 4, refer to section “[Channel Initiator - trace costs](#)”.

To help understand the data generated by accounting and statistics trace, see SupportPac [MP1B](#) “Interpreting accounting and statistics data”.

Accounting Trace Costs

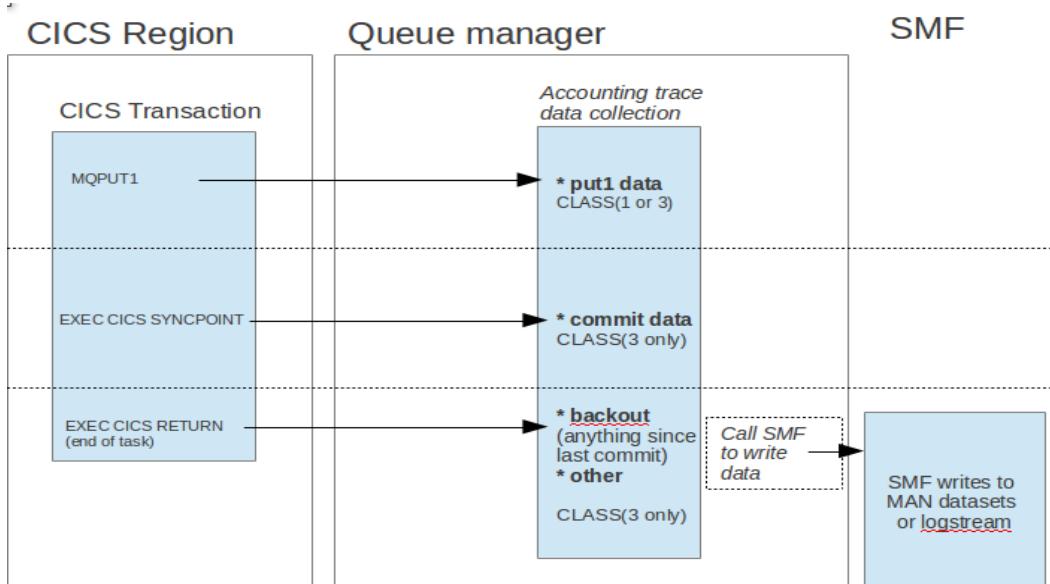
Accounting trace classes 1 and 3 write an SMF 116 record for each transaction.

Additional “continuation” records may be written when class 3 accounting is enabled depending on the number of queues accessed by the transaction. Typically the primary SMF 116 record can support up to 8 queues and each continuation record can support a further 9 queues. For example, an application that accesses 20 queues would see 3 SMF 116 records, one primary and 2 continuation records, the first of which has data for 9 queues and the second has data for the final 3 queues.

The amount of data written to SMF can impact the CPU costs when enabling accounting trace, particularly in an environment with a high transaction rate, as can where the SMF data is stored, for example we have observed that writing high volumes of data to logstreams² is lower cost than writing to SMF MAN datasets.

In a CICS environment with a high transaction rate, the logging of records to SMF MAN datasets may result in the queue manager reporting CSQW133E ‘TRACE DATA LOST’ messages as SMF may not be able to write the data to its datasets sufficiently quickly. In this case it is advisable either to use logstreams rather than SMF MAN datasets or to use TRACE(A) CLASS(3) for short periods of time (60 seconds).

Consider a CICS transaction that performs a single MQPUT1 followed by an EXEC CICS SYNCPOINT and EXEC CICS RETURN when the connected queue manager has accounting trace enabled:



The diagram attempts to show when the queue manager will gather accounting data.

²SMF write rates are discussed in section “IBM MQ and zEnterprise Data Compression (zEDC)”

- Accounting data will be stored for the MQPUT1 for either class 1 or 3.
- When trace class 3 is enabled:
 - EXEC CICS SYNCPOINT will result in the queue manager storing “commit” data.
 - EXEC CICS RETURN will result in the queue manager storing “backout” and “other” data relating to the end of task processing. **Note:** The action performed under the IBM MQ accounting data type “backout” is not actually backing out, rather a resetting of the transaction state within the queue manager.
- At transaction end, the queue manager calls SMF to request the data be written to the SMF data repository (MAN datasets or logstreams).

Storage Usage

Prior to MQ version 8, MQ would initialise certain accounting data areas regardless of the class 3 accounting trace status. Enabling class 3 accounting would cause further data areas to be allocated as well as driving the writing of the data to SMF.

From MQ version 8 onwards, the storage used for accounting is only allocated when class 3 accounting is enabled.

The total storage used per connection for accounting trace class 3 is usually between 4KB and 8KB but can be higher when a connection accesses many queues.

Who pays for data collection?

Typically there is an increase in the costs attributed to the application address space.

Class 3 accounting may also see an increase in the costs attributed to the queue manager address space.

Who pays for writing to SMF?

There is typically an increase in CPU usage in the SMF address space, which is dependent on the amount of data written in each SMF 116 record.

Class 3 accounting may also see an increase in the queue manager address space for the aggregation of task related data.

How much data is written?

Accounting trace class 1 typically writes 436 bytes per transaction, regardless of the number of queues accessed by the transaction.

Accounting trace class 3 data will depend on the number of queues used but as a guide, a transaction accessing 2 queues typically writes 8324 bytes.

A class 3 SMF 116 record with 8 queues would typically write 25076 bytes.

This means that for storing data for 1,000,000 transactions the following storage would be required:

- class 1: 415MB
- class 3 with 2 queues: 7938MB
- class 3 with 8 queues: 23914MB

Trace(A) CLASS(1) costs: TRACE(A) CLASS(1) can be estimated as:

- ***Data gathering:***
 - 2 microseconds per API (MQPUT, MQPUT1, MQGET only)
- ***Writing to SMF:***
 - plus 1-5 microseconds. The record is fixed length and is relatively small keeping costs down.

Trace(A) CLASS(3) costs: The costs for TRACE(A) CLASS(3) can be estimated as³:

- ***Data gathering:***
 - 1-3 microseconds per API (including commit, backout and other)
- ***Writing to SMF:***
 - plus 6-50 microseconds for the primary SMF record (depending on size), lower costs were observed when fewer queues were accessed)
 - plus up to 60% of the cost of writing the primary SMF record for each continuation record, again depending on how many queues the continuation record contains.

³Based on measurements on zEC12.

Comparing costs – a working example: Consider a CICS transaction that costs 1 millisecond (1000 microseconds) and performs the following interactions with an MQ queue manager:

MQPUT1	
COMMIT	as a result of EXEC CICS SYNCPOINT
BACKOUT	as a result of EXEC CICS RETURN
OTHER	as a result of EXEC CICS RETURN

Enabling **class 1 accounting trace** would be expected to add the following cost to the transaction:

MQPUT1	+2 microseconds.
Writing to SMF	+3 microseconds.
Total	1005 microseconds (+0.5%)

Enabling **class 3 accounting trace** would be expected to add the following cost to the transaction:

MQPUT1	+2 microseconds.
COMMIT	+2 microseconds.
BACKOUT	+2 microseconds.
OTHER	+2 microseconds.
Writing to SMF	+25 microseconds

Total	1033 microseconds (+3.3%)
-------	---------------------------

Accounting trace considerations

- The impact of accounting trace will depend on what proportion of the transaction is weighted towards MQ – for example if a transaction spends 50% of its lifetime in MQ, the impact of trace may be much higher than those in the preceding examples.
- By contrast a transaction that performs DB2 SQL, reading and writing of files and complicated calculations may see a small impact from accounting trace.
- The size and persistence of the message being put or gotten does not impact the cost of accounting trace.
- Class 3 accounting in a high transaction environment can generate vast amounts of data. It can be useful to enable this trace periodically to monitor your systems.

Statistics Trace Costs

TRACE(S) costs are not significant as they are not dependent on transaction rate nor transaction volumes.

This includes the queue statistics trace, class(5), that was introduced in [MQ for z/OS 9.3](#).

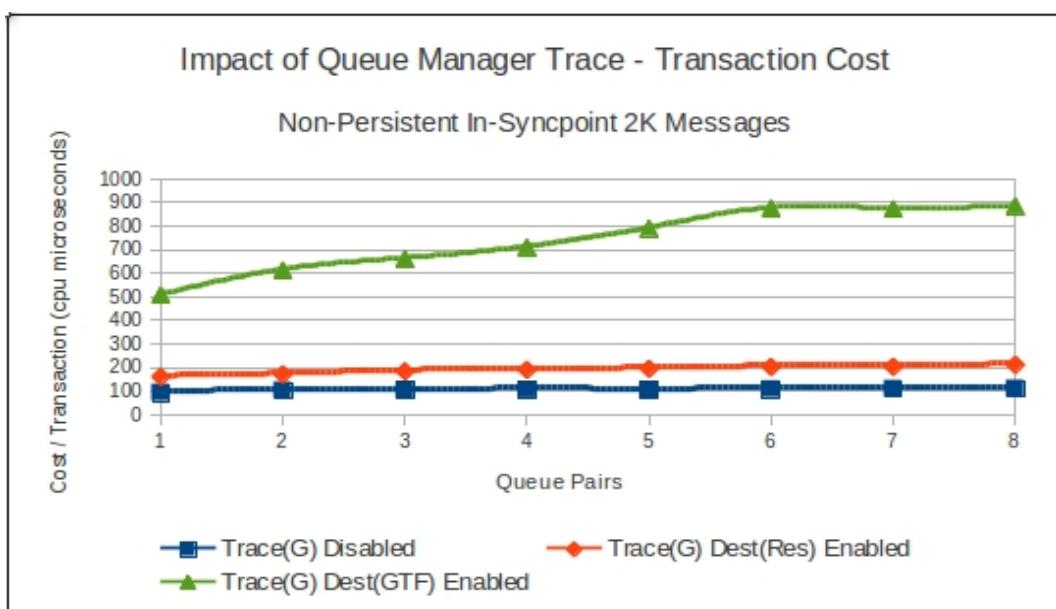
Global Trace Costs

NOTE: From a performance perspective, disabling TRACE(G) is advised as the global trace costs vary significantly depending on the MQ workload being processed.

WebSphere MQ for z/OS version 7.1.0 changed how the queue manager global trace is gathered. In previous releases, the global trace data was stored in a single storage area which on a busy system with multiple processors could result in a high degree of contention when writing the trace data. Version 7.1.0 exploits 64-bit storage to allocate an area of storage for each thread, which reduces the contention issues seen previously.

The destination of the trace can have a significant affect on the performance achieved when running with trace enabled.

The following 2 charts show the impact of running with TRACE(G) enabled in an LPAR with high workload, low resource contention from an MQ perspective, i.e. workload is spread out over multiple queues that are hosted on multiple pagesets with multiple buffer pools, only a single pair of requester/server applications accessing each pair of queues.

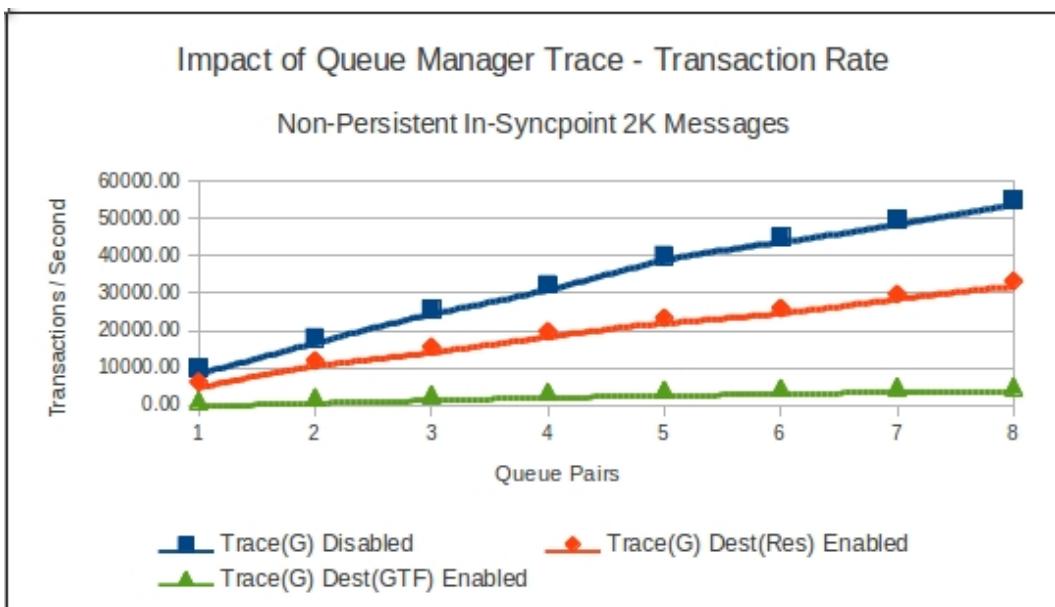


The first chart compares the cost per transaction when attempting to process messages in a request/reply type scenario where the server messages are processed inside of syncpoint, i.e. the server gets and puts its messages in-syncpoint.

NOTE: This transaction cost is the cost incurred by the requester application, the server application and the queue manager address space. The applications used are performing very little non-IBM MQ related workload.

- When TRACE(G) is off, the transaction cost is relatively flat at approximately 100 microseconds.
- When TRACE(G) with DEST(RES) enabled, the effect of global trace doubles the transaction cost.
- With TRACE(G) DEST(GTF) the transaction cost increases significantly to more than 8 times the cost of running with global trace disabled.

The subsequent chart shows how the achieved transaction rate is affected by global trace.



The above chart shows how the achieved transaction rate are constrained much sooner when TRACE(G) is enabled, especially with destination GTF selected. A significant proportion of the time is spent waiting for serialization to process the trace records.

Performance / Scalability

Maximum throughput using persistent messages

You should consider whether you really need persistent messages. Many applications do not require the advantages of persistent messages, and can make use of the cheaper, faster non-persistent messages instead. Some users do not use persistent messages at all!

If you use persistent messages then allocate your log data sets on your best DASD.

What factors affect persistent message throughput ?

The extra factor affecting throughput of persistent rather than non-persistent messages is the performance of the IBM MQ log, which depends on the following:

- Type and overall usage of the DASD subsystem used for the MQ log data sets.
- The data rate that the DASD subsystem can sustain for the IBM MQ log data sets. This sets the upper limit for the IBM MQ system.
- The DASD subsystem and control unit type, the amount of DASD cache, and the number of channel paths to the DASD. All will have an effect on throughput.
- Total I/O contention within the system.
- IBM MQ log data set placement. Having logs on heavily used DASD volumes can reduce throughput.
- The average time to complete an I/O to the DASD volume, which depends on the amount of data to be logged as well as the DASD subsystem characteristics and IBM MQ log data set placement. Using zHPF may provide some relief in this instance – see “[Benefits of using zHPF with IBM MQ](#)”

See “[Maximum persistent message throughput – private queue](#)” for an example of the rates achieved.

Application syncpoint specifics

- The rate of commits or out of syncpoint requests. This is application specific. Each commit or out of syncpoint request involving a persistent message requires the application to wait for completion of an MQ log I/O (a log force). A two-phase commit requires the application to wait for the completion of two separate MQ log I/Os (2 log forces).
- The worst case is 2 MQ log forces for each MQPUT and each MQGET. For instance.
 - Consider a CICS application might update a recoverable VSAM file and MQPUT a message in one unit of work. This requires a 2-phase commit and therefore 2 MQ log forces.
 - That message is then processed (MQGET) by a DB2 stored procedure which updates a database and commits the unit of work. This requires a 2-phase commit coordinated via RRS and therefore 2 MQ log forces.
- Another case is 1 log force for each MQPUT and each MQGET. For instance consider
 - A channel receiving messages at an achieved batchsize of 1, MQPUTs each message and commits the batch (this is typical for a channel unless deliberate batching of messages or a very high message rate occurs). This requires a 1-phase commit.
 - A CICS program MQGETs this message, updates a DB2 database, MQPUTs a reply message then commits. This requires a 2-phase commit.

- A channel MQGETs and sends the reply message back to the originator at an achieved batchsize of 1. This requires a 1-phase commit.
- Thus there are 4 MQ log forces for the 2 messages processed, which is an average of 1 log force per MQPUT and MQGET.
- Because each application using persistent messages is likely to be I/O-bound on the log you will probably need many application instances to achieve best throughput.
- However, some applications require strict message ordering. This means only a single application instance is allowable.

Message size and number of messages per commit

Message size and number per commit affect the amount of data which must be written to the log for each log force.

Similar amounts of data per commit will usually give similar throughput. For example, 5 persistent messages of size 1KB require about 11.5KB of data to the log when fully processed by MQPUT and MQGET. 1 persistent message of size 10KB requires a similar amount of log data. Similarly 50 persistent messages of size 1KB which are MQPUT in one unit of work and MQGET in one unit of work will have similar IBM MQ log performance as one persistent message of 100KB.

Indexed Queues

Indexed queue considerations

If a non-indexed queue contains more than a few messages and an MQGET with a specific MSGID or CORRELID is used then costs can be very high as the queue will have to be searched sequentially for the matching message. Clearly any queue used by an application that requires specific MQGETs should be specified with the appropriate INDXTYPE.

Prior to version 7.0.1, queue indexes were maintained in 31-bit queue manager storage. This meant that there was an implementation limit as to how many messages could be stored on an indexed queue and on our systems this was around 7.2 million messages. From version 7.0.1 of WebSphere MQ for z/OS onwards, indexed queue data is maintained in 64-bit storage and the queue manager is able to store in excess of 100 million messages on indexed queues.

The amount of storage used for each message on an indexed queue is 136 bytes of above bar storage.

These indexes must be recreated during queue manager initialization for all persistent messages in each indexed private queue. This requires that the first page of all the messages for each indexed queue be read from the pagesets. This is done sequentially queue by queue. For private indexed queues this will increase initialization elapsed time by the order of a few milliseconds per page read. For instance, a private indexed queue consisting of 8 million persistent messages increases elapsed time of initialization by about 250 seconds using DS8950 DASD.

QSGDISP(SHARED) indexed queues have indexes implemented within the CF list structure and so do not require recreation at queue manager initialization. The maximum number of messages in a QSGDISP(SHARED) indexed queue is limited only by the maximum number of messages possible in a CF list structure.

Private indexed queue rebuild at restart

Private indexed queues have virtual storage indexes which must be rebuilt when a queue manager restarts. IBM MQ allows these indexes to be rebuilt in parallel and offer the “[QINDBLD \(WAIT/NOWAIT\)](#)” CSQ6SYSP parameter. The WAIT option gives previous release behaviour and is the default whereas, NOWAIT allows initialization to complete before the index rebuilds complete.

Thus NOWAIT allows all applications to start earlier. If an application attempts to use an indexed queue before that queue’s index is rebuilt then it will have to wait for the rebuild to complete. If the rebuild has not yet started then the application will cause the rebuild to start immediately, in parallel with any other rebuild, and will incur the CPU cost of that rebuild.

Each index rebuild still requires that the first page of all the messages for that indexed queue be read from the page set. The elapsed time to do this is of the order of a few milliseconds per page read. Buffer pool page usage is not significantly affected by the index rebuild. Thus other applications will not be impacted by buffer pool contention with index rebuilds.

Up to ten separate index rebuilds can be processed in parallel plus any rebuilds initiated by applications.

How long will it take to restart a queue manager with deep indexed local queues?

When a queue manager is restarted and there are persistent message on the indexed queues, it is necessary for the queue manager to rebuild those indexes.

This rebuilding process can be seen in the queue manager log as below:

```
CSQI007I @QMGR CSQIRBLD BUILDING IN-STORAGE INDEX FOR <queueName>
CSQI006I @QMGR CSQIRBLD COMPLETED IN-STORAGE INDEX FOR <queueName>
```

As mentioned previously the queue manager is not available for work until all the indices are rebuilt, unless CSQ6SYSP parameter QINDXBBLD is set to NOWAIT.

The depth of the indexed queue impacts the time taken to restart a queue manager.

The queue manager allocates a maximum of 10 threads to rebuild indexes.

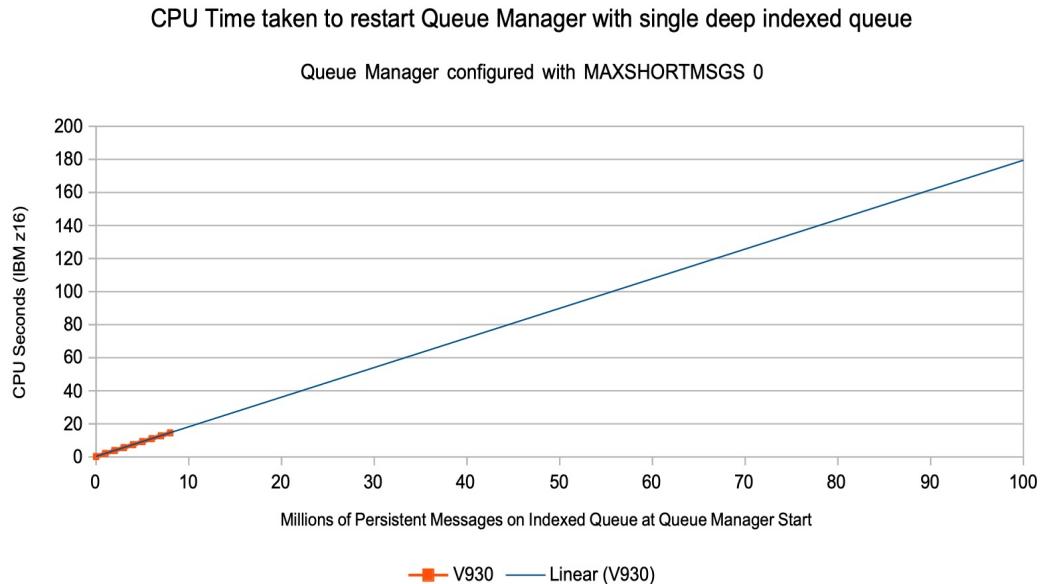
- If durable subscriptions exist, the SYSTEM.DURABLE.SUBSCRIBER.QUEUE may be rebuilt first.
- The deepest indexed queues are rebuilt next.

To be able to index a queue, each message has to be read:

- For short messages where MAXSHORTMSGS 0 has been set, multiple messages may exist on a single page
- For other messages, there will be one page read for each message.
- For deep queues there will be significant page set activity.

The effect of a single deep indexed queue upon Queue Manager restart

The following chart shows the measured CPU cost to start an MQ for z/OS 9.3 queue manager when a single indexed queue has increasing depth. The queue manager has been configured with MAXSHORTMSGS 0 to allow up to 8 messages of 100 bytes per 4K page.



A trend line has been added to provide an indication of how long it would take to restart a queue manager with 100 million small messages.

In the measurements for a single deep queue, the index rebuilding process was able to use a single processor at approximately 10% usage.

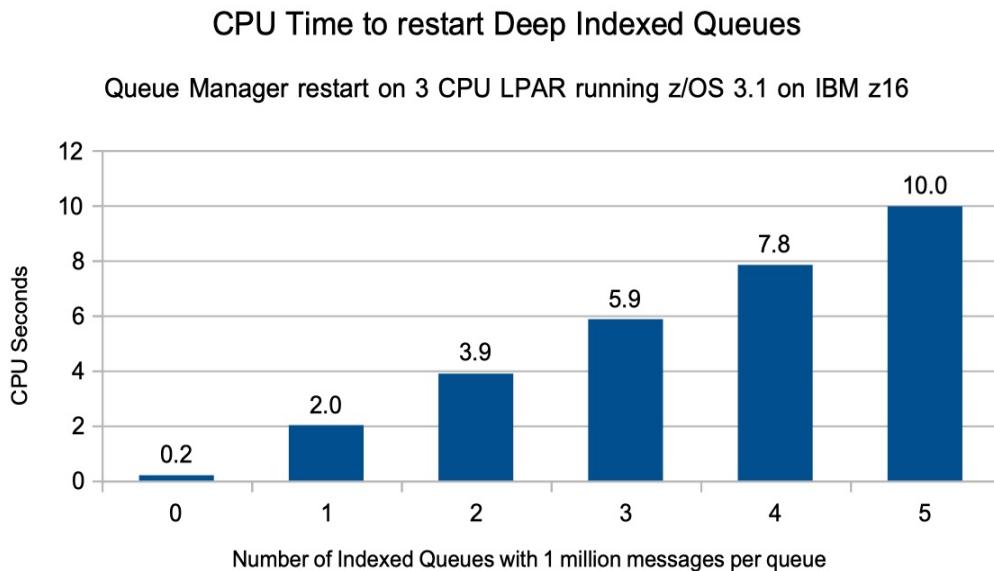
This means that for a queue with 8 million short messages on the queue, the CPU time was 14.5 seconds and the elapsed time was 147 seconds.

Using the trend line, we would predict that a queue with 100 million messages would use 180 CPU seconds and would take approximately 30 minutes to restart on a 3 processor LPAR of an IBM z16.

The effect of a multiple deep indexed queues upon Queue Manager restart

The following measurement shows the measured CPU cost to restart a queue manager with an increasing number of deep indexed queues on a single page set.

If the queue manager had not been configured with MAXSHORTMSGS 0, each 64GB pageset would have been limited to 16 million messages of up to 3697 bytes.



As with the single deep indexed queue, the rebuilding of multiple indices is not a particularly CPU-intensive function, so the measurement is not constrained by having 3 processors available when re-building more than 3 queues concurrently. In this case the time taken to rebuilt the queue indices is constrained by the rate at which the data can be read from DASD.

In terms of elapsed time taken to restart the queue manager, with a single queue that had 1 million messages, the queue manager was ready after 19 seconds. When restarting with 5 queues each with 1 million messages, the queue manager was ready after 20 seconds and the system reported peak CPU usage to the equivalent of 75% of a single processor.

Note: For the measurements with a single deep indexed queue and multiple deep indexed queues, the z/OS LPAR was configured with zHPF disabled. When the measurements were repeated with zHPF, the elapsed time for the re-building of the indices was reduced by up to 33%.

Queue manager initiated expiry processing

If the queue manager attribute EXPRYINT is non-zero then at startup and subsequent EXPRYINT second intervals any messages whose expiry time has been passed will be deleted by a system process. EXPRYINT can be changed, including to or from zero using an ALTER QMGR command. The default for EXPRYINT is zero, which gives the previous release behaviour of no queue manager initiated expiry processing. Minimum non-zero EXPRYINT is 5 seconds.

The “[REFRESH QMGR TYPE\(EXPIRY\) NAME\(.....\)](#)” command requests that the queue manager performs an expired message scan for every queue that matches the selection criteria specified by the NAME parameter. (The scan is performed regardless of the setting of the EXPRYINT queue manager attribute.)

For private local queues this system process uses significantly less CPU time than employing your own application to browse complete queues. This is partly because the system knows when there is no possibility of there being any expired messages on a private local queue and because if it is necessary to browse a queue, the system process avoids the overheads involved in repeated calls across the application/system boundary. For the case where the system knows there are no messages to expire on any private queue the CPU cost at each scan is not significant.

For shared local queues each defined queue must be processed. A single queue manager, of those with non-zero EXPRYINT in the queue sharing group, will take responsibility for this processing. If that queue manager fails or is stopped or has its EXPRYINT set to zero then another queue manager with non-zero EXPRYINT will takeover. The CPU cost at each EXPRYINT interval will depend on a number of factors:

- Number of messages on queue (all messages including those with expiry not set will be scanned) as the message may be put by a different queue manager in the QSG).
- Size of message on queue
- Where the message is stored

For example, for 1KB messages on a shared queue, the cost is of the order 5 CPU microseconds (8561-703) per each message. This cost increases to 10 CPU microseconds (8651-703) when the system actually browses and deletes the expired messages.

With a non-zero EXPRYINT in a queue sharing group, it is worth emphasising that all messages on shared queues will be scanned - each of which requires a call to the CF. If the message is determined to be eligible for expiry, a second call to the CF will be made. On our IBM z15 with an internal CF, all calls were synchronous and took an average 3.5 CPU microseconds of CF CPU. A less responsive CF, whether remote, duplexed or generally busier may take longer to perform the scan and expiry calls.

If the message has been offloaded to DB2 or shared message data sets, the cost may be higher. For example, when 1KB messages are offloaded to shared message data sets, the cost to determine whether the message can be expired is double that of a message stored solely in the coupling facility.

The time to browse a queue and delete any expired messages will be significantly less than using your own equivalent application because this system process avoids the overheads involved in repeated calls across the application / system boundary.

Queue manager security

How much storage is used?

When a IBM MQ queue manager on z/OS is started and the security switch profiles have been defined such that user-ids need to be authenticated, there is a cost to the queue manager to use this authentication information.

WebSphere MQ for z/OS version 7.0.1 introduced the use of 64-bit storage for holding security information relating to IBM MQ. All storage used by the security manager is now held in 64-bit storage, which means that the ECSA usage does not increase as more user IDs or IBM MQ resources are used. As a result, the number of user IDs that can access IBM MQ resources is limited by the amount of auxiliary storage.

The environment being measured

The measurements were run on a system configured as shown below:

- A 2097 with 8 CPUs available running z/OS 1.11
- A version 7.0.1 WebSphere MQ queue manager with security enabled.
- A CICS region running CTS 3.2
- A number of transactions have been created e.g.
 - A transaction ran an application to put a 1K non-persistent message to a named queue and then get the message from the same queue.
- WebSphere Studio Workload Simulator for z/OS (formerly known as “TPNS”) was used to drive a workload through the CICS environment.
 - TPNS scripts were created to log onto CICS using security and then run a number of transactions using between 1 and 50 MQ queues. This process was repeated for a range of user-ids.

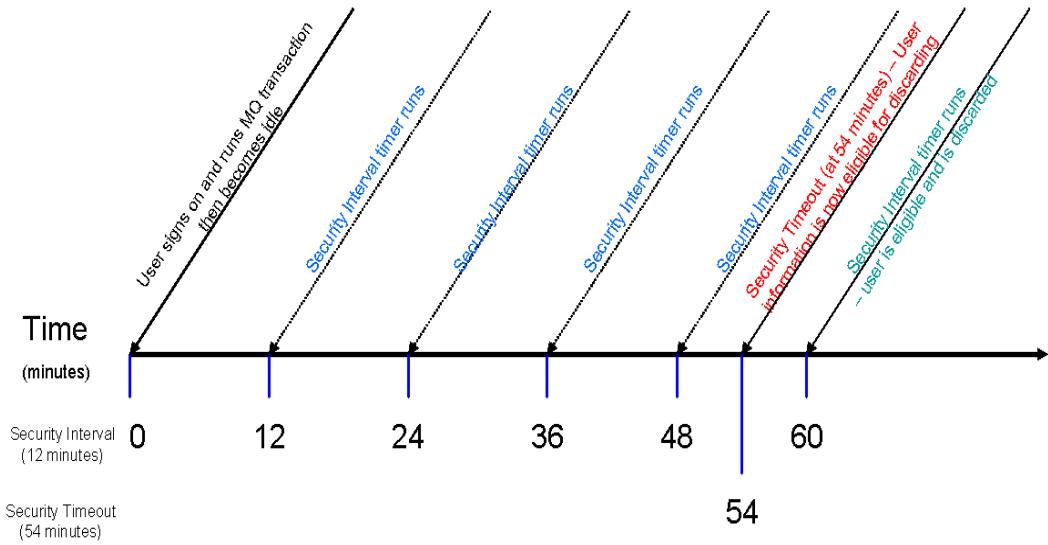
Issuing a “DISPLAY SECURITY” command against the queue manager shows:

```
CSQH015I MQPZ Security timeout = 54 minutes
CSQH016I MQPZ Security interval = 12 minutes
CSQH030I MQPZ Security switches ...
CSQH034I MQPZ SUBSYSTEM: ON, 'MQPZ.NO.SUBSYS.SECURITY' not found
CSQH034I MQPZ CONNECTION: ON, 'MQPZ.NO.CONNECT.CHECKS' not found
CSQH034I MQPZ COMMAND: ON, 'MQPZ.NO.CMD.CHECKS' not found
CSQH034I MQPZ CONTEXT: ON, 'MQPZ.NO.CONTEXT.CHECKS' not found
CSQH034I MQPZ ALTERNATE USER: ON, 'MQPZ.NO.ALTERNATE.USER.CHECKS' not found
CSQH034I MQPZ PROCESS: ON, 'MQPZ.NO.PROCESS.CHECKS' not found
CSQH034I MQPZ NAMELIST: ON, 'MQPZ.NO.NLIST.CHECKS' not found
CSQH034I MQPZ QUEUE: ON, 'MQPZ.NO.QUEUE.CHECKS' not found
CSQH034I MQPZ COMMAND RESOURCES: ON, 'MQPZ.NO.CMD.RESC.CHECKS' not found
CSQH022I MQPZ CSQHPDTC 'DIS SEC' NORMAL COMPLETION
```

The “security timeout” refers to the number of minutes from last use that the information about a user ID is retained by IBM MQ.

The “security interval” is the time that passes between an MQ process checking the last use time of all authenticated user IDs to determine whether the security timeout period has passed. If the timeout period for a user is exceeded, the information is discarded by the queue manager.

The following time-line attempts to indicate when security-related timer processing will be invoked.



The chart shows that when the security interval is 12 minutes, there is a timer running every 12 minutes to determine whether there are any unused user IDs that are eligible for discarding.

It is not until 54 minutes after the user has signed on and completed their last IBM MQ transaction that they are eligible for discarding.

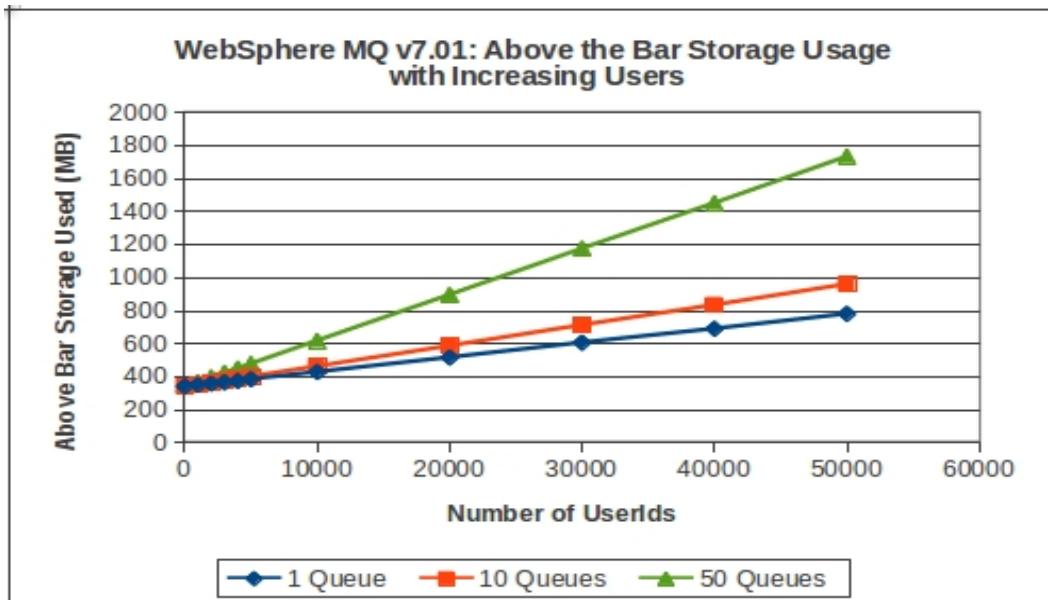
Since the interval runs every 12 minutes, there is a period (in this example) of 6 minutes where the user id is eligible for discarding but is not discarded.

At 60 minutes after completing their MQ transaction, the users' information is discarded from MQ.

From version 7.0.1 of Websphere MQ for z/OS, the Security Manager uses a pooling principle so that when the users' information is discarded, the storage is returned to a pool for subsequent re-use.

The data

The following chart shows the amount of additional virtual storage required when security has been enabled on the queue manager.



Notes on chart:

1. The line titled “1 Queue” is where the TPNS script will run
 - CICS Sign-on followed by 1 transaction putting and getting a non-persistent message for each user id (up to 50,000 unique user ids).
2. The line titled “10 Queues” is where the TPNS script runs:
 - CICS Sign-on followed by 10 serialised transactions putting and getting non-persistent messages from separate queues for each user id (from 1 to 50,000 unique user ids).
3. The line titled “50 Queues” is where the TPNS script runs:
 - CICS Sign-on followed by 50 serialised transactions putting and getting non-persistent messages from separate queues for each user id (from 1 to 50,000 unique user ids).

What can we gather from the chart?

When the user is just issuing a sign-on to CICS followed by a single transaction involving MQ, there is an associated cost of approximately 8.8KB per user of auxiliary storage which includes storage for 1 queue. This is approximate since the 64-bit storage is allocated 1MB blocks.

When the user performs other MQ transactions that affect further queues, additional security information is cached so the storage usage increases. For example when the users workload affects 50 MQ queues, there is a cost of 28.2KB per user of auxiliary storage; an increase of 19.4KB per user.

This means that there is a base cost per user of 8.80KB (which includes accessing 1 queue) when running with security enabled as per the display security command shown previously. Additionally there is a 405 byte cost for each subsequent queue that the user hits as part of their work.

Virtual storage usage

When migrating IBM MQ on z/OS from version 7.x to version 9.3.0 the private storage usage is similar. This section shows the usage and gives some actions that can be taken to reduce storage usage.

From version 8.0.0, the following enhancements relate to storage:

- 64-bit storage used for:
 - Buffer pools (optionally)

From version 7.1.0, the following enhancements relate to storage:

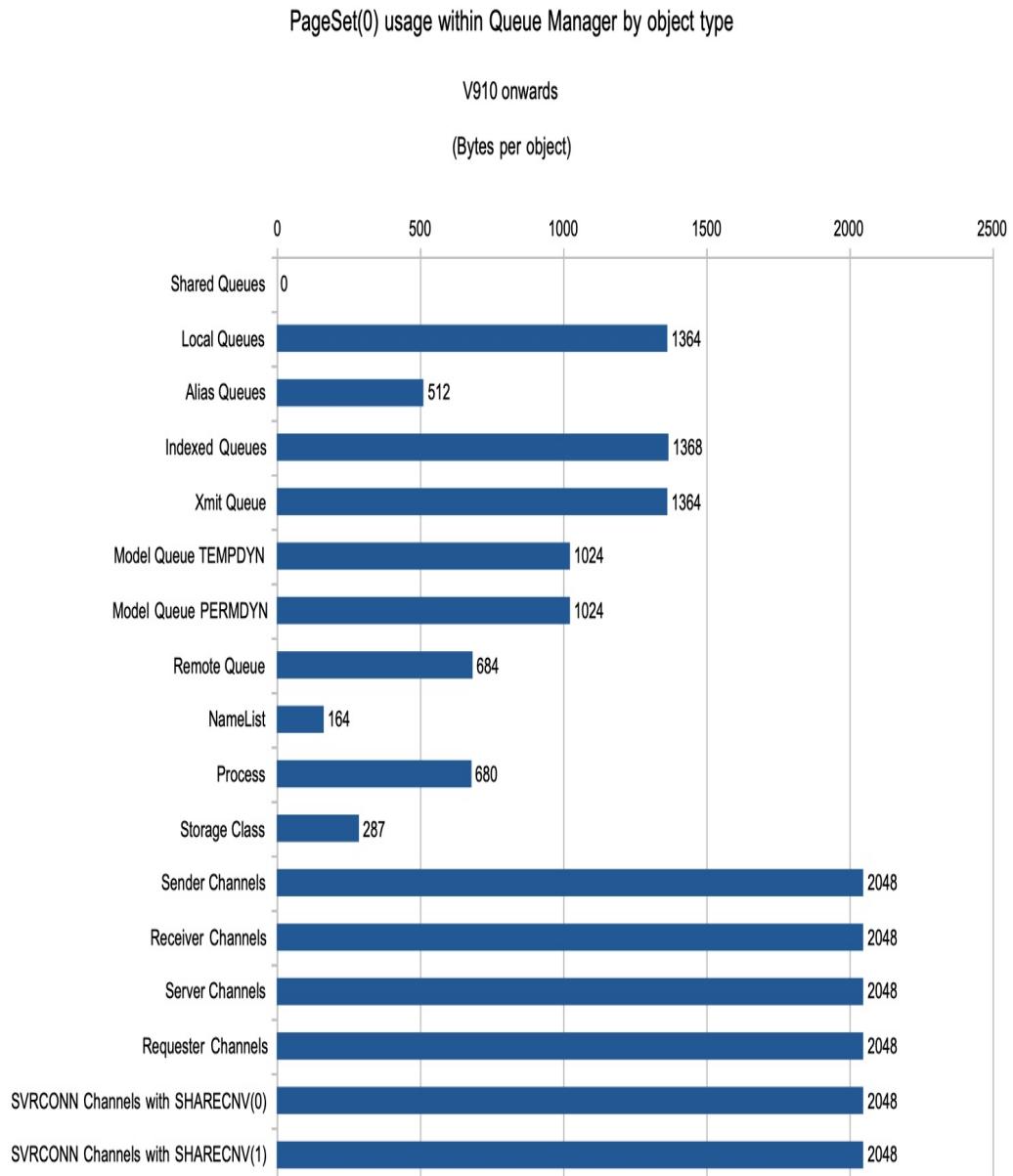
- 64-bit storage used for:
 - Topic Manager
 - Security Manager
 - Indexed Queues
 - Intra-Group Queuing Buffer
 - CFLEVEL(5) shared message data set (SMDS) offload capability
 - CHLAUTH cache

Object sizes

When defining objects the queue manager may store information about that object in pageset 0 and may also require storage taken from the queue manager's extended private storage allocation.

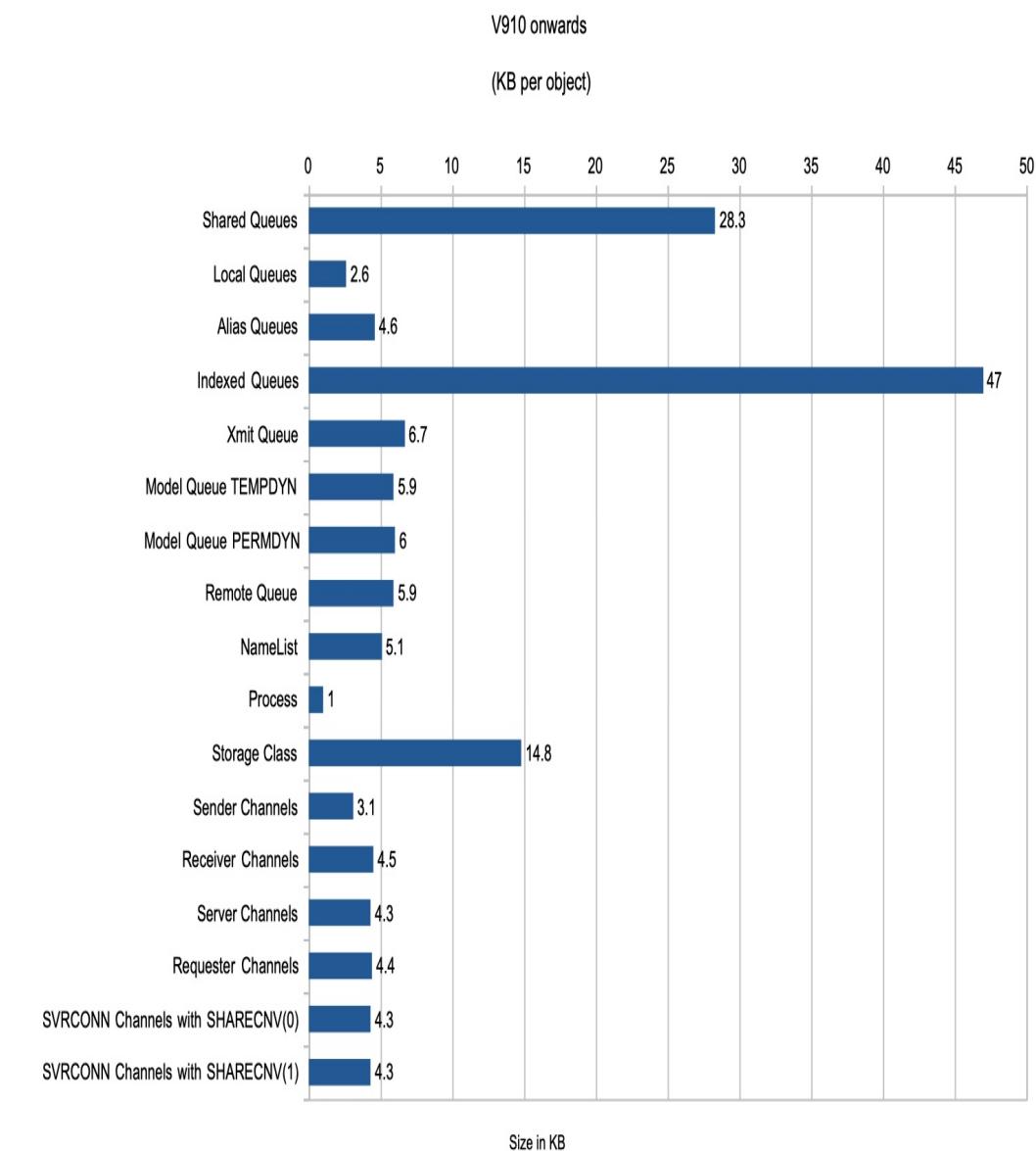
The data shown in the following 2 charts only includes the storage used when defining the objects.

Page set 0 usage



Virtual storage usage by object type

Extended Private Storage usage within Queue Manager by object type



NOTE: CHLAUTH objects are cached in 64-bit storage.

Initial CSA (and ECSA) usage

CSA usage is similar between V710 to V930 when similarly configured queue managers are started. On our systems this is approximately 6MB per queue manager.

CSA usage per connection

CSA usage has seen little change in the following releases: V710 through to V930:

- For local connections, MCA channels and SVRCONN channels with SHARECNV(0), CSA usage is 2.47KB per connection.
- For SVRCONN channels with SHARECNV(1), CSA usage is approximately 4.9KB per connection.
- For SVRCONN channels with SHARECNV(5), CSA usage is approximately 3KB per connection (based on 5 clients sharing a channel instance).
- For SVRCONN channels with SHARECNV(10), CSA usage is approximately 2.7KB per connection (based on 10 clients sharing a channel instance).

Buffer Pool Usage

Typically the buffer pools have used the most virtual storage in the queue manager. Version 8.0.0 allows the buffer pools to be defined using 64-bit storage. This means that there is more space available in the queue manager's 31-bit storage for other things that haven't been moved above the bar, e.g. more handles.

Version 8.0.0 also allows more buffer pools such that a 1:1 mapping with the page sets is possible, allowing a more granular level of control of the sizing of buffer pools according to the usage.

Provided sufficient 64-bit storage is available, you may be able to define the buffer pools as large as the associated pageset.

If storage is limited, you should have a large buffer pool for the short-lived messages, where short-lived is typically a few minutes duration, and a small buffer pool for long-lived messages.

You may be able to reduce the size of buffer pools and maintain good performance, but your performance may degrade if you have an unexpected spike in message throughput which causes your buffer pool to fill up and have less than 15% free buffers. You can monitor the QPSTCBSL fields in the QPST statistics for this.

Storage for security information

From version 7.0.1 Security Manager was changed to use 64-bit storage to hold the security information. Additionally, the storage obtained is not released until queue manager termination, rather the storage used following a user ID being discarded is returned to a storage pool for subsequent re-use. When queue level security is enabled:

- Each user-id which accesses the queue manager requires about 8.8KB of auxiliary storage. Typically 4.66KB of the 8.8KB are "backed" in real storage.
- For each permanent queue that a user uses, then on average it uses 405 bytes per queue.

As all security storage is held in 64-bit storage, the queue manager should not be constrained for storage due to the number of user IDs held.

It is still advisable to set the Queue Manager Security keywords Interval and Time-out to limit the duration that user ID information is cached in the queue manager. The Time-out value specifies how long an un-used user ID can remain in the queue manager. The default time is 54 minutes.

Reducing this time will cause unused information to be discarded, returning the storage to the pool. The interval is the duration between checking to see if the time out value has expired.

Impact of number of objects defined

At startup all of the object information is loaded into buffer pool zero. If this buffer pool is large enough to contain all of the objects, then the control blocks for the object stay resident. If buffer pool zero is too small then unused objects will be removed from the buffer pool (but will still be present on disk, and will be read into the buffer pool if the object is used).

The average storage used for local queues is about 2660 bytes per object. This was determined from the storage increase when 10,000 local queues were defined, divided by 10,000. This figures included wasted space in a 4K page when the objects do not fit perfectly.

If you are constrained for storage you can decrease the size of buffer pool to have enough space for the active objects + 20%. Once the system has started and warmed up, then there should be few pages read from the page set.

Use of indexed queues

When a queue is defined as being indexed then additional control blocks are created to define the index. For each queue with indxtype specified, 47KB of memory are required.

From WebSphere MQ version 7.0.1, the index data for an indexed queue is stored above the 2GB bar. This removes the constraint within the queue manager relating to the depth of an indexed queue over a non-indexed queue.

Object handles

When an application has a queue open, then a handle control block is allocated within the queue manager. This uses about 4K of memory per handle. If there are many concurrent applications, or applications have a large number of open queues then this can lead to a large number of handles.

For example if you have 10,000 client applications, and each client application gets from a queue and puts to a reply queue, then there will be 20,000 handles using 80MB of virtual storage. An MCA channel can have up to 30 queues open at a time, for example when messages from a remote queue manager are being sent to different queues. With 1000 channels, this could be up to 30,000 handles or 120MB, though typically a channel only puts to a few queues.

Number of pages in use for internal locks

Locks are taken by the queue manager to serialize usage of the data and resources. Large messages or large number of messages in a unit of work can lead to a large number of locks being used.

You can reduce the storage required by ensuring that the applications are well behaved:

- Do not process large number of messages in a unit of work. You can use the queue manager attribute MAXUMSGS to limit the number of messages processed in a unit of work.
- Process small numbers of large messages in a unit of work. When moving messages over channels, you might consider using channel attribute **BATCHLIM** to restrict the size of the unit of work for large messages without impacting the batch size when messages are smaller.. Fast (non-persistent) messages are processed out of syncpoint, so the locks are released when the put has completed.

Shared queue

There is increased memory use when a queue manager is used in a QSG. When a queue manager is configured to be part of a Queue Sharing Group, it uses an additional 27MB of storage in the private region.

For each application structure that is used, the queue manager uses 800KB of virtual storage in the private region, 2KB of ESQA and 4KB of ECSA.

Using BACKUP CFSTRUCT command

The BACKUP CFSTRUCT command allocates 65MB. This remains allocated until the system detects that the queue manager is short of virtual storage, and releases any unused storage.

Clustering

If you are using clustering then information about the status of clustering is held both in the channel initiator and the queue manager. The queue manager has two views of the cluster cache, one in key 7, for the queue manager, and one in key 8 for the cluster workload manager applications.

The minimum cache size is 2MB (or 4MB for both copies). The cache size will be calculated dynamically from the configuration, rounded up to the nearest MB and have 1MB extra added. If your configuration changes significantly then this cache can fill up. If using a static cache, the queue manager must be restarted to extend the cache size. If using dynamic cache, the cache will be extended automatically and dynamically (no queue manager restart is required).

Chapter 2

Coupling Facility

CF link type and shared queue performance

Shared queue performance is significantly influenced by the speed of Coupling Facility (CF) links employed. There are several different types of CF links. These include (see the result of a ‘D CF’ operator command).

Link Type		Maximum Operating Rate	Distance
CFP	Coupling Facility Peer Or InterSystem Channel-3 (ISC-3)	200 MB/second.	Up to 20KM unrepeated. Maximum 100KM. Not available on z13 onwards.
PSIFB	Parallel Sysplex coupling over InfiniBand	Rate varies on hardware and link type, e.g. 5 GB/second on z13 using 12x IFB3 1 GB/second on z13 using 12x IFB.	Up to 150 metres. Not available on z15 onwards.
CS5	Coupling Short distance 5	z13-z16: 8 GB/second.	Maximum 150 metres..
CL5	Coupling Long distance 5	z13-z16: 800 MB per second.	Up to 10KM unrepeated. Up to 100KM with qualified DWDMs (Dense Wavelength Division Multiplexer).
ICP	Internal Coupling Facility Peer	Fastest connectivity using memory to memory data transfers.	Internal speeds, z14, z15 and z16.

NOTE: Further information on coupling links can be found in section “Link Technologies” in document [“Coupling Facility Configuration Options”](#).

All link types can be present. The operating system generally selects the fastest currently available. Some uses of the CF are normally synchronous in that they involve what is effectively a very long instruction on the operating system image while a function request and its associated data are passed over the link, processed in the CF, and a response returned. CPU time for synchronous CF calls is thus to some extent dependent on the speed of the CF link. The slower the link the higher the CPU cost. The faster the CPU the higher the equivalent instruction cost.

System z has heuristics which allow it to change potentially synchronous CF calls to asynchronous. This limits potentially rising CPU costs but can affect throughput as more CF calls become asynchronous.

CF processor time is also dependent on the speed of the CF link, but much less so than the operating system.

All these factors can make prediction of shared queue performance on other systems based on results in this document less accurate than for private queue.

Faster links will generally improve CPU cost and throughput.

How many CF structures should be defined?

A minimum of two CF structures are required. One is the CSQ_ADMIN structure used by IBM MQ. All other structures are application structures used for storing shared queue messages.

Up to 512 shared queues can be defined in each application structure. We have seen no significant performance effect when using a large number of queues in a single application structure.

We also have seen no significant performance effect when using multiple structures rather than fewer larger structures.

If few large application structures are used, the queues would be able to be deeper, meaning there is less likelihood of receiving an MQRC 2192 “storage medium full” message.

If using many smaller application structures, fewer queues and potentially less applications will be affected should one of the queues gets to such a high depth that an MQRC 2192 is reported.

Typically we recommend using as few CF application structures as possible, but there may be situations where it is advisable to put queues that may be deep into separate structures where an MQRC 2192 will not affect mission-critical queues.

If all the MQGETS and MQPUTS in an application are out of syncpoint, the cost of using one or more application structures remains the same.

If any MQPUTS and MQGETS are within syncpoint, a single CF application structure is recommended. For the locally driven request/reply workload using a single queue manager but with the server input queue in a different application structure to the common reply queue the unit CPU cost per request/reply increased by 6% for non-persistent messages and 12% for persistent messages. This also resulted in a decrease in throughput of 5% for non-persistent messages and 2% for persistent messages.

What size CF structures should be defined?

What values for MINSIZE, INITSIZE, and SIZE (maximum size) should be used and should ALLOWAUTOALT(YES) be specified in the CFRM (Coupling Facility Resource Manager) policy definition for IBM MQ CF structures?

- Consider making SIZE double INITSIZE.
- Consider making MINSIZE equal to INITSIZE, particularly if ALLOWAUTOALT(YES) is specified.
- It is recommended to define SIZE to be not more than double INITSIZE. The value of SIZE is used by the system to pre-allocate certain control storage in case that size is ever attained. A high SIZE to INITSIZE ratio could effectively waste a significant amount of CF storage.

If the entire CF reaches an installation-defined or defaulted-to percent full threshold as determined by structure full monitoring, the system will consider reducing the size of any structures with unused space that have been defined with ALLOWAUTOALT(YES).

For this reason we advise consideration of making MINSIZE equal to INITSIZE so that IBM MQ structures will not be made too small. This is particularly important for the CSQ_ADMIN structure which could cause failure of shared queue operations if it becomes too small.

CSQ_ADMIN

This CF structure does not contain messages and is not sensitive to the number or size of messages but it is sensitive to the number of queue managers in the QSG.

Version 7.1 updated the supplied sample member SCSQPROC(CSQ4CFRM) to specify INITSIZE of 20000KB for the example CSQ_ADMIN structure. This should be sufficiently large to allow 7 queue managers to connect to the QSG up to CFCC levels 24. For CFCC level 25, both the INITSIZE and SIZE attributes will need to be increased by 10000KB to support the same number of queue managers.

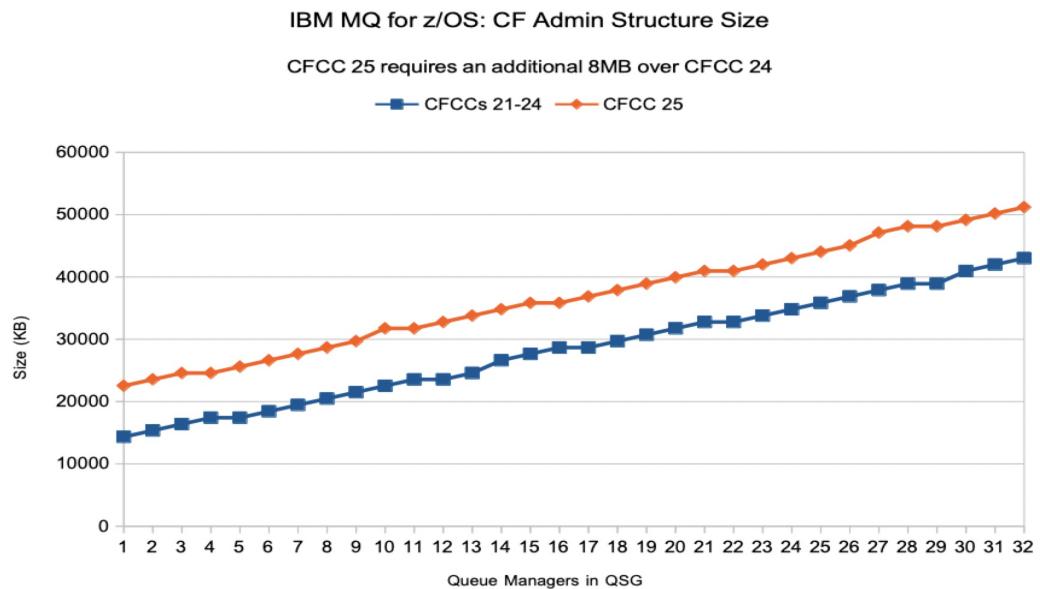
The IBM MQ command “[DIS CFSTATUS\(CSQ_ADMIN\)](#)” shows the maximum number of entries in this structure, for instance ENTSMAX(20638) on a CF at CFCC level 24. This command also shows the current number of entries used, for instance ENTSUSED(47). A queue manager in a queue sharing group is only allowed to start if there are at least 1000 entries available per started queue manager. So our example is adequate for at least 7 queue managers in a QSG using a CF at CFCC level 24. Each successive CF level tends to need slightly more control storage for the CF’s own purposes, so ENTSMAX is likely to decrease each time your CF level is upgraded.

CSQ_ADMIN usage is affected by the number of messages in each unit of work, but only for the duration of the commit for each UOW. This only need be a concern for extremely large UOWs as the minimum size structure is enough for a UOW of about 40,000 messages. This is larger than the default maximum size UOW of 10,000, defined by MAXUMSGS.

The use of UOWs with very large numbers of messages is NOT recommended. Where large units of work are being used in conjunction with shared queues, the CSQE038E “Admin structure is full” message may be logged when there is insufficient space in the structure for the unit of work. This may be followed by the queue manager terminating with a 5C6-00C53002 abend.

How large does my admin structure need to be?

The size of the admin structure depends on the number of queue managers in your queue sharing group. The following chart shows the required size of the admin structure by number of queue managers in the QSG.



Application structures

IBM MQ messages in shared queues occupy storage in one or more pre-defined CF list structures. We refer to these as application structures to distinguish them from the CSQ_ADMIN structure. To estimate an application structure size:

Use the IBM [CF Sizer for MQ](#) tool or the following algorithm may be used:

- Estimate typical message sizes (including all headers except the MQMD)
- For each typical message size
 - If <= 63KB (64512 bytes) then:
 - Add the 372 bytes for implementation headers (including MQMD v1 header).
 - If using message grouping add an additional 72 bytes for the MQMDE.
 - Round up to 256 byte boundary (subject to a minimum of 1536 bytes).
 - Add 256 bytes for the CF Structure ENTRY (1 for each message)
 - Multiply by maximum number of messages.
 - Normally messages will reside in a IBM MQ application structure only long enough for the exploiting application to retrieve them. However if the exploiting application suffers an outage that prevents it from retrieving messages from the structure, the structure must be large enough to retain any messages that might be written to it, for the duration of the outage. You must therefore consider:
 1. The number of queues that map to the structure,
 2. The average put rate for each queue (i.e. the rate at which messages are written to the structure),
 3. The maximum outage duration to be tolerated.
 - Add to total for all typical message sizes
 - Add 32%¹ for CFCC level 12 and above (for other implementation considerations, this percentage can be much greater for application structures smaller than 16MB). Previous CFCC levels required the addition of 25%.
 - Add 8MB for CFCC level 25.
- If typical message size > 63KB (64512 bytes) then
 - For CF application structures with ALLOWAUTOALT(NO) allow about 2KB of CF storage per message larger than 63KB.
 - CF application structures with ALLOWAUTOALT(YES) will eventually have an ENTRY to ELEMENT ratio reflecting the average for all messages. This is difficult to estimate but it is usually sensible to also use this 2KB per message estimate.
However, consider the special case of all messages being larger than 63KB. The CF storage usage for shared queue messages larger than 63KB is 1 ENTRY and 2 ELEMENTS per message. This means that the actual requirement is about 1KB per message. ALLOWAUTOALT(YES) structures will eventually adjust themselves such that 1 million such messages (all larger than 63KB) would require about 1GB of CF storage.

Use this result for INITSIZE in the operating system CFRM policy definition. Consider using a larger value for SIZE in the CFRM policy definition to allow for future expansion. See “[Increasing the maximum number of messages within a structure](#)”.

¹Some of this 32% is to maintain the 1:6 entry to element ratio and some is CF overhead.

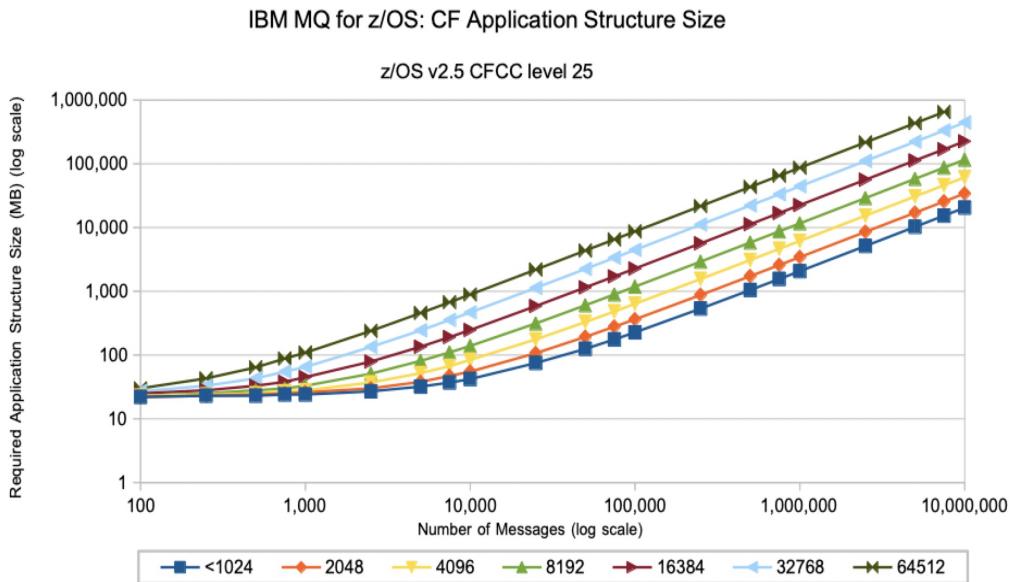
The following CFRM policy definition of an approximately 0.5GB CF list structure is typical of those used for our measurements.

```
STRUCTURE NAME(PRF2APPLICATION1)      /* PRF2 is the QSG name */
SIZE(1000000)
INITSIZE(500000)
PREFLIST(SOCF01)
```

See the IBM MQ Planning guide for details of MQ definitions.

How many messages fit in a particular CF application structure size?

To get some idea of how many messages you can get for a particular CF application structure size consult the following chart where ‘message size’ includes the user data and all the headers except the MQMD.



NOTE: The chart uses log scales. For instance the tick marks between 1000 and 10000 on the x axis are the 2000, 3000, 4000 and so on up to 9000 messages. The tick marks between 1 and 10 on the y axis are 2, 3 and so on up to 9 MB of required structure size.

For example, you can get about:

- 600 messages of 64512 bytes (63KB) in a 64MB structure,
- Or nearly 50000 16KB messages in a 1GB structure.

A CF at levels prior to CFCC level 12 will accommodate a few percent more messages than this chart, but only up to a 4GB limit.

A CF at level CFCC 17-24 requires the structure to be approximately 5MB larger to store an equivalent number of messages to a CF at level CFCC 14.

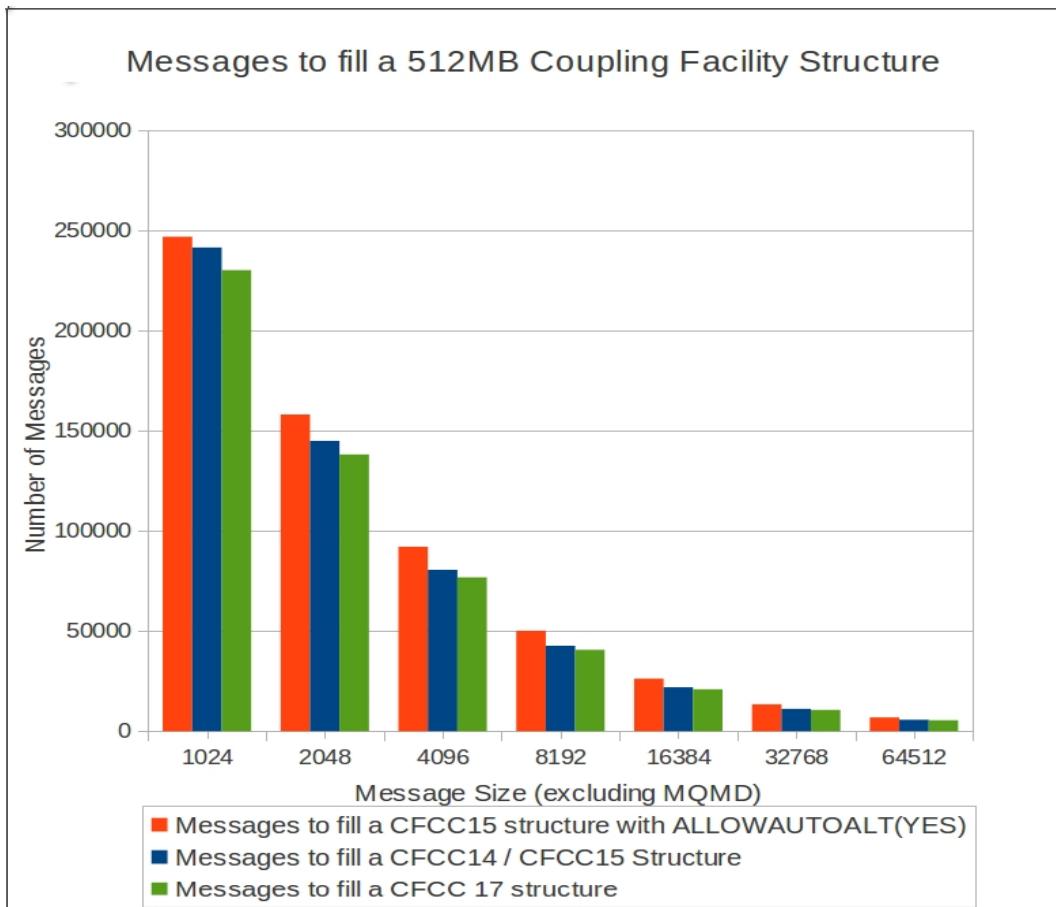
A CF at level CFCC 25 requires the structure to be approximately 8MB larger to store an equivalent number of messages to a CF at level CFCC 15-24.

CF at CFCC levels 17 and later

The following table gives approximate message capacity of a IBM MQ CF application structure sized at 0.5GB, assuming it is defined in the CFRM policy as ALLOWAUTOALT(NO).

Post-CFCC level 14, CFCC levels use more storage, reducing capacity somewhat, although this has stabilised for CFCC 17 through 24 e.g.:

Message size (excluding only MQMD)	Approximate messages in 0.5GB Structure	
	CF at CFCC level 17-24	CF at CFCC level 25
All message sizes <= 1164	230,000	227,000
2,048	138,000	137,000
4,096	76,700	75,000
8,192	40,600	39,600
16,384	21,200	21,000
32,768	10,900	10,800
64,512	5,600	5,600



Sizing structures at CFLEVEL(5)

CFLEVEL(5) provides the ability to increase the capacity of the CF by implementing a 3-tiered offload procedure.

Implementing tiered thresholds allows higher capacity whilst not penalising performance until the CF resource becomes constrained.

By default, the CFLEVEL(5) structure will offload messages greater than 63KB to the shared message data set.

In addition, there are 3 default thresholds:

1. Offload all messages larger than 32KB (including headers) when the structure is 70% full.
2. Offload all messages larger than 4KB (including headers) when the structure is 80% full
3. Offload all messages when the structure is 90% full.

Example: Consider a scenario where only 16KB messages are used with a 0.5GB structure.

CFLEVEL(4)

A 16KB message would require 1 entry and 66 elements.

A 0.5GB structure would support approximately 24,000 messages, with ALLOWAUTOALT=YES.

CFLEVEL(5)

16KB messages stored in the CF would still require 1 entry and 66 elements, whereas all offloaded messages use 1 entry and 2 elements.

For example, 16KB messages would not be offloaded until the structure reaches 80% full. This means that 17,101 messages are stored in their entirety in the CF. Upon reaching the 80% threshold, remaining messages are offloaded to the SMDS datasets. Provided the SMDS datasets are large enough, the CF would then be able to store a total of 158,166 messages.

Increasing the maximum number of messages within a structure

The maximum number of messages can be increased dynamically either by:

1. Increasing the size of a structure within the currently defined CFRM policy limits. This can be done by operator command or by the system for structures defined ALLOWAUTOALT(YES).
2. Using CFLEVEL(5) structures which implement tiered thresholds.
3. Changing the ENTRY to ELEMENT ratio, which can be done only by the system and only to a structure which is defined in the CFRM policy with ALLOWAUTOALT(YES)

The ELEMENT to ENTRY ratio is initially fixed by IBM MQ at 6 to 1. The system then pre-allocates ELEMENTS and ENTRIES in that ratio to fill the INITSIZE of that structure (having reserved space for its own control information including being able to cope with this structure at its maximum possible size (SIZE)).

NOTE: A structure is full if either all ENTRYs or all ELEMENTs are in use.

Every message requires an ENTRY and enough ELEMENTs to contain all message data and headers. Each ELEMENT is of size 256 bytes. Now consider the ELEMENT and ENTRY requirement for various message sizes, remembering to add the 372 bytes, that covers the implementation headers (including the MQMD v1 header), to each message.

For example,

- 5000 byte message requires 21 ELEMENTS and 1 ENTRY
- 300 byte messages require 3 ELEMENTS and 1 ENTRY
- 10 byte messages require 2 ELEMENTS and 1 ENTRY

Taking the above sizing and applying them to a simple scenario can show how we achieve the 6 to 1 ratio.

No. of Messages	Size (bytes)	ELEMENTS	ENTRY	Maintaining 6:1 ratio means:
1	5000	21	1	Unused 3 ENTRIES
1	300	3	1	Unused 3 ELEMENTS
3	10	6	3	Unused 12 ELEMENTS
Total (5)		30	5	Unused: 15 ELEMENTS 3 ENTRIES

So for the above example, we have achieved the 6 to 1 ratio, although we have lost 15 ELEMENTS and 3 ENTRIES.

If we continue to add only 5000 byte messages, we will run out of ELEMENTS long before the ENTRIES are used.

Alternatively, if we add only 10 or 300 byte messages, we will run out of ENTRIES long before we run out of ELEMENTS.

System initiated alter processing is the only way to adjust ENTRY to ELEMENT ratio for IBM MQ CF structures. It can also change the size of a CF list structure up to the maximum (SIZE) or down to the minimum defined (MINSIZE) as defined for that structure.

To see ENTRY and ELEMENT information use z/OS command "D XCF", for example:

```
D XCF,STR,STRNAME=PRF2APPLICATION1.
```

Use of system initiated alter processing

This facility allows the system to alter the size of a structure (both up and down) and to change the ENTRY to ELEMENT ratio.

The following CF list structure definition is possible for application CFSTRUCT named APPLICATION1 in queue sharing group PRF2:

```
STRUCTURE NAME(PRF2APPLICATION1)
SIZE(1000000)      /* size can be increased by z/OS      */
INITSIZE(500000)    /* from 500000K to 1000000K by        */
MINSIZE(500000)    /* or decreased to 500000K by        */
ALLOWAUTOALT(YES)   /* system initiated ALTER processing */
FULLTHRESHOLD(80)
PREFLIST(SOCFO1)
```

When the FULLTHRESHOLD is crossed the operating system will take steps to make adjustments to the list structure ENTRY to ELEMENT ratio to allow more messages to be held within the current size, if possible. It will also, if necessary, increase the size towards the maximum (the value of SIZE). This process is not disruptive to ongoing work provided there are sufficient processors available in the Coupling Facility. However, it can take up to several minutes after the threshold is crossed before any action is taken. This means that a structure full condition, IBM MQ return code 2192, could easily occur before any such action is taken.

For structures containing predominantly message sizes less than 908 bytes (5 * 256 - implementation headers (372)) and greater than 63KB (64512 bytes) then it is likely that considerably more messages can be accommodated in the same size structure after any such adjustment.

To reiterate, if the entire CF reaches an installation-defined or defaulted-to percent full threshold as determined by structure full monitoring, the system will consider reducing the size of any structures with unused space that have been defined with ALLOWAUTOALT(YES).

For this reason we advise consideration of making MINSIZE equal to INITSIZE so that IBM MQ structures will not be made too small. This is particularly important for the CSQ_ADMIN structure which could cause failure of shared queue operations if it becomes too small (queue manager failure prior to V6).

User initiated alter processing

The following system command is an example of how to increase the size of a structure:

```
SETXCF START,ALTER,STRNAME=PRF2APPLICATION1,SIZE=750000
```

This command increases the size of the structure but does not change the ENTRY to ELEMENT ratio within the structure. Increasing CF structure size is not noticeably disruptive to performance in our experience.

Decreasing CF structure size is not advised with CFCC levels prior to level 12 as there are circumstances where it is very disruptive to performance for a considerable time.

How often should CF structures be backed up?

Highly available parallel sysplex systems often have stringent recovery time requirements. If you use persistent messages in any particular application structure it will need to be backed up.

If backup is infrequent then recovery time could be very long and involve reading many active and archive logs back to the time of last backup. Alternatively an application structure can be recovered

to empty with a “**RECOVER CFSTRUCT(..) TYPE(PURGE)**” command, but this does mean that any messages on the queues defined to the structure being purged will be lost.

The time to achieve a recovery is highly dependent on workload characteristics and the DASD performance for the log data sets of individual systems. However, you can probably aim to do backups at intervals greater than or equal to the desired recovery time.

CF application structure fuzzy backups are written to the log of the queue manager on which the BACKUP command is issued. The overhead to do a backup is often not significant as the number of messages in an application structure is often not large. The overhead to do a backup of 200,000 1KB persistent messages is less than 0.5 CPU seconds on a 8561-703 system.

When the 1KB persistent messages were stored on SMDS, the backup costs did not increase but the rate at which the backup completed dropped from 290MB/sec to 13MB/sec.

The recovery processing time is made up of the time to:

- Restore the fuzzy backup of the CF structure, which is typically seconds rather than minutes.
- Re-apply the net CF structure changes by replaying all log data, including non-shared queue work, written since the last fuzzy backup.

The logs of each of the queue managers in the queue-sharing group are read backwards in parallel. Thus the reading of the log containing the most data since fuzzy backup will normally determine the replay time.

The data rate when reading the log backwards is typically less than the maximum write log data rate. However, it is not usual to write to any log at the maximum rate it can sustain. It will usually be possible and desirable to spread the persistent message activity and hence the log write load reasonably evenly across the queue managers in a queue sharing group. If the actual log write data rate to the busiest queue manager does not exceed the maximum data rate for reading the log backwards then the backup interval required is greater than or equal to the desired recovery time.

Backup frequency example calculation

Consider a requirement to achieve a recovery processing time of say 30 minutes, excluding any reaction to problem time. As an example, using DS8900F DASD with the queue manager doing backup and restore on a 8561-703 system running z/OS V2R5, we can restore 200,000 1KB persistent messages from a fuzzy backup on an active log in 2 seconds. To meet the recovery processing target time of 30 minutes, we have more than 29 minutes to replay the log with the most data written since the last fuzzy backup. The maximum rate at which we can read an active log data set backwards is about 450MB/sec on this system, so we can read about 760GB of the longest log in 29 minutes.

When data needed to be read from archive logs, the read rate dropped to approximately 60MB/second, so it is worth re-iterating that for best performance, there are sufficient active logs to be able to recover from backups.

The following table shows the estimated backup interval required on this example system for a range of message rates:

1KB persistent msgs/sec to longest log	1KB persistent msgs/sec to 3 evenly loaded logs	MB/sec to longest log	Backup interval in hours (based on reading logs backwards at 450MB/Sec)
1000	3000	2.27	95
2000	6000	4.61	46
38000	114000	112	1.9
(38,000 is the maximum for this DASD with 1KB messages)			

A crude estimate for the amount of log data per message processed (put and then got) by queue managers in a QSG is message length plus 1.33KB.

Backup CFSTRUCT limit

When calculating how frequently to back up your CF structures, it is important to consider how much data is to be backed up.

With the capacity of SMDS, it is possible to exceed MQ recovery limits, and as such it is worth considering the following variables:

Variable	Values	Description
MAXARCH	10-1000	Maximum number of archive logs that can be recorded in the BSDS. When the number is exceeded, recording begins again at the start of the BSDS.
Maximum log size	4GB	This means the maximum recoverable backup can be 4000GB (3.9TB). The most recent data may be recovered from active logs. For example if the queue manager is defined with 310 active logs there could be:- <ul style="list-style-type: none">• 1 current active log• 309 active logs with data that is also in the most recent 309 archive logs• 691 archive logs with data not in active logs.
Maximum size of single SMDS	16TB	MQ supports 63 application structures and 32 queue managers in the QSG. This means that each structure could have 512TB in SMDS and the QSG could contain 30 PB of SMDS data.

Even when the volume of data being backed by the BACKUP CFSTRUCT command exceeds the maximum configured storage available for backup (`((MAXARCH * log size) + (log size * number of active logs))`), the backup will appear to work.

It is only when attempting to recover the data that problems may occur due to the size of the backup. You may see the following messages when structure recovery is attempted and fails:

- CSQE132I Structure recovery started, using log range from LRSN=xxxx to LRSN=yyyy
- CSQJ113E RBA zzzz NOT IN ANY ACTIVE OR ARCHIVE LOG DATA SET..
- CSQE112E Unable to recover structure APPLICATION1, failed to read required logs

Administration only queue manager

If there ever might be a lot of persistent messages or a lot of persistent message data to be backed up then the normal persistent message workload could be impacted while the log of the queue manager doing the backup is extra busy.

If this is a serious potential concern then consider defining an extra queue manager in the QSG and use it only for administration purposes such as [BACKUP CFSTRUCT\(..\)](#).

When should CF list structure duplexing be used?

CF list structure duplexing gives increased availability at a performance cost.

Any version of MQ that supports shared queues can be used with duplexed CF structures without change to either the code or the MQ definitions.

Availability within a given QSG may be summarised as follows:

SIMPLEX CF Structure definition	Action on single failure
CSQ_ADMIN V6 (or later) queue managers	<p>Queue managers stay up and rebuild this structure from their logs.</p> <p>Only serialised applications need to wait for rebuild completion.</p> <p>Rebuild only completes when every queue manager defined in the QSG has done its work. This means that if a queue manager was down at the time of failure it must be restarted before any new serialised applications can start unless the queue managers are V7.0.1 or later.</p> <p>Note: Shared channels are serialised applications.</p> <p>V7.0.1 saw the introduction of peer admin rebuild.</p>
CSQ_ADMIN V5 queue managers	<p>Entire QSG fails.</p> <p>The structure is rebuilt from logs at restart. All queue managers in the QSG need to restart to complete the rebuild.</p> <p>Only serialised applications need to wait for rebuild completion.</p>
Application structure CFLEVEL(1)	<p>ALL currently connected queue managers fail.</p> <p>On restart the structure is reallocated, all messages are lost</p>
Application structure CFLEVEL(3 or higher)	<p>No queue manager fails.</p> <p>Applications using queues in that structure fail.</p> <p>On restart persistent messages can be recovered by any queue manager in the QSG provided that any queue manager in the QSG has done a backup and all subsequent logs are available.</p> <p>Alternatively the structure can be ‘recovered’ to empty.</p>

DUPLEX CF Structure definition	Action on single failure
CSQ_ADMIN	Entire QSG remains available, z/OS recovers to duplex.
Application structure CFLEVEL(1)	ALL currently connected queue managers remain available, z/OS recovers to duplex.
Application structure CFLEVEL(3 or higher)	ALL currently connected queue managers remain available, z/OS recovers to duplex.

How does use of duplexed CF structures affect performance of MQ?

MQ operations on CF structures are typically nearly all update operations. Duplexed CF structure updates incur significant extra CF CPU and link usage. The following guidelines assume that there will be adequate total resources available. An overloaded CF is likely to cause significant performance problems for the entire sysplex.

Estimating performance for duplexed versus simplex CF structures is complex and even more than usually workload and system configuration dependent for the following reasons.

CPU costs

The CPU cost impact of duplexed CF structure compared to simplex CF structure usage depends on the link types used both between the z/OS image and the two CF's being used as well as the link between the two CF's that are being used for duplexing.

Note that one of these two CF's might have changed after a structure failure and recovery back to duplex and thus performance characteristics might also change after recovery.

Operations that update the contents of a CF structure have more impact on extra CPU cost than those that do not. MQPUTs and destructive MQGET's clearly have to update the CF structure containing the message and MQCMITs have to update the CSQ_ADMIN structure. An MQGET for browse causes no updates.

Throughput

Throughput for shared queue non-persistent messages, even when kept on duplexed CF structures, is always going to be much better than for any sort of persistent message because of the elapsed time required for DASD logging I/O necessary to provide media recovery for persistent messages.

Throughput for messages on a duplexed CF structure compared to a simplex CF structure is impacted by the type of links used between the z/OS image and the two CF's and by the type of links between the two CF's.

Any throughput impacts of duplexing CF structures are because:

- Update operations are asynchronous for duplexed CF's. They may be synchronous or asynchronous between z/OS and the CF for simplex CF structures, depending on operating system heuristic decisions.
- The operation can only complete at the speed of the slowest link.
- The second CF may be physically much more distant (possibly many kilometres and even light takes time to travel). 10KM distance will add something of the order of 200 - 250 microseconds to the service time for each request.

CF Utilization (CF CPU)

The CF utilization cost will increase significantly for MQ update operations when using duplex rather than simplex CF list structures.

- Each of the duplexed CF's must process the operation
- Plus there is synchronization between the CF's.

The CF utilization for MQ update operations on the CF of the primary copy structure will approximately double. The secondary copy CF utilization will be nearly as much as the primary.

Environment used for comparing Simplex versus Duplex CF structures

1 LPAR on a 2084 with 3 dedicated processors, rated as a 2084-303, with ICP and CFP links to local coupling facility. ICP link (fastest) will be used when available.

Coupling facility has 3 engines available.

Physically the duplexed structures are located locally but only CFP (ISC-3) links between the 2 coupling facilities.

Multiple Queue Sharing Groups defined:

- One with duplexed CSQ_ADMIN structure and 3 application structures of which one is duplexed.
- All other QSGs have simplex CSQ_ADMIN structure and 3 application structures of which one is duplexed.

Locally driven request / reply workload with multiple requester applications putting to a common input queue and getting a reply message by CORRELID from a separate common indexed queue. Multiple server applications getting from the common input queue and putting to the reply-to queue.

Duplexing the CSQ_ADMIN structure

From observations on our system when running our locally driven request / reply workload we have derived the following general guidance.

Note: The test system has a multiple links from the LPAR to the primary CF, including a fast ICP link which will be used when available, and a slow link from the primary CF to the secondary (duplexed) CF.

- CPU cost between 0% and 30% greater.
- For non-persistent messages processed in-syncpoint, throughput decreases by 30% for 1KB and 20% for 32KB messages.
- For persistent messages processed in-syncpoint, throughput decreases by 25% for 1KB and 12% for 32KB messages.
- The contribution of CSQ_ADMIN structure usage to CF utilization is usually much less than that for the application structures. Duplexing the CSQ_ADMIN structure might typically increase the MQ caused load by 10% for 63KB non-persistent messages to 33% for 1KB non-persistent messages.
- The use of messages contained in more than one application structure within a unit of work increases the activity to the CSQ_ADMIN structure and so would further increase CPU and CF utilization and decrease throughput.

Using a slower CFP link from the LPAR to the primary CF and another CFP link from the primary CF to the duplexed CF:

- CPU cost between 3 and 8% greater
- For non-persistent messages processed in-syncpoint, throughput decreases by 18% for messages between 1 and 63KB.

When using the faster ICP link between the LPAR and the primary CF with a slower CFP link from the primary CF to the secondary CF, the additional cost of duplexing the admin structure is significantly more than when running with a CFP link from the LPAR to the CF. Despite this, the faster ICP link does allow up to 40% more throughput for 1 to 63KB non-persistent messages that are processed in-syncpoint.

Duplexing an application structure

It really only makes sense to duplex an application structure if the CSQ_ADMIN structure is also duplexed. From our observations on our system with our locally driven request /reply workload we have derived the following general guidance for duplexing of both CSQ_ADMIN and the application structure.

NOTE: The test system has a multiple links from the LPAR to the primary CF, including a fast ICP link which will be used when available, and a slow link from the primary CF to the secondary (duplexed) CF.

- CPU cost about 15% greater for 1KB persistent messages and 30% greater for 1KB non-persistent messages.
- CPU cost about 12% greater for 32KB persistent messages and 25% greater for 32KB non-persistent messages.
- Throughput decrease by 40% for 1KB non-persistent messages and 50% for 10KB non-persistent messages. For persistent messages throughput decrease is negligible for 1KB messages and rises to less than 10% for 32KB messages. The use of messages contained in more than one application structure within a unit of work increases the activity to the CSQ_ADMIN structure and so would further decrease throughput.
- The contribution of MQ CF structure usage to CF utilization will double for the primary structures. The secondary structures will use almost as much as the primary.

Non persistent shared queue message availability

Non-persistent messages are not logged whether in private or shared queues. Therefore they cannot be recovered if lost. Nevertheless, shared queue non-persistent messages have much greater availability than private queue non-persistent messages.

Private queue non-persistent messages are lost when the queue manager fails or shuts down normally. Even with simplex CF structure usage shared queue non-persistent messages are not easily lost. They are only lost if the CF application structure containing them fails or is deleted by the operator. In particular, they are NOT lost when any or even all queue managers in a queue sharing group fail or shut down normally (except failure caused by loss of that application structure).

Users may consider using non persistent shared queue messages, with all the advantages of pull workload balancing which come with use of shared queue, where they might previously have required persistent messages in a non-shared queue environment. In this case there is generally a CPU cost saving and potentially a significant increase in throughput compared to use of non-shared queue persistent messages.

Existing users of private queue persistent messages moving to shared queue non-persistent messages on CF structures may see a CPU cost saving and potentially a significant increase in throughput even when using duplexed CF structures.

Coupling Facility

What is the impact of having insufficient CPU in the Coupling Facility?

As the coupling facility becomes more utilized, the system will convert synchronous requests to asynchronous requests. The system has heuristic algorithms which decide that the system will be more efficient if it issues an asynchronous request with the cost of a re-dispatch, rather than have the processor wait for a synchronous request to complete. With the asynchronous request, the processor can process other work.

With a highly utilized coupling facility, a small increase in utilization can result in significant increase in coupling facility response time when the CF requests are changed from synchronous to asynchronous.

By example:

- When the CF was 20% utilized, the majority of the CF requests were synchronous taking 4.4 microseconds.
- When the CF was 70% utilized, the majority of the CF requests were synchronous taking approximately 9.9 microseconds, but those asynchronous requests were taking 600 microseconds.

When do I need to add more engines to my Coupling Facility?

On a coupling facility with 1 engine, it is advised that when the %busy value exceeds 30% for any period of time, an additional engine should be added. On a coupling facility with multiple engines, it is advised that when the %busy value (as can be seen from an RMF III ‘‘CF Activity’’ report) exceeds 60%, an additional engine should be added.

What type of engine should be used in my Coupling Facility?

The CF can use general CP’s that can also be used by the operating system LPAR or an ICF, which can only run CFCC. For performance reasons, the ICF engines do not share L3 and L4 caches with z/OS processors.

IBM recommends that *dedicated engines should always be used for a CF whose response times are critical.*

If dedicated processors are not available, review “[Coupling Facility Configuration Options](#)” which recommends the use of the dynamic dispatch “DYNDISP” configuration option when using shared CPs.

CF Level 19 - Thin Interrupts

With the introduction of CFCC level 19, the options for sharing CF processors increased such that the Dynamic CF dispatching (DYNDISP) option now supports ON, OFF and THIN.

The performance of THIN interrupts is discussed in detail in the white paper “[Coupling Thin Interrupts and CF Performance in Shared Processor Environments](#)”.

CF Level 25 - Thin Interrupts

CFCC level 25 deprecates the DYNDISP=ON | OFF options . As of CFCC level 25, shared engine CF dispatching uses DYNDISP=THIN.

Why do I see many re-drives in the statistics report?

What is a re-drive?

The Coupling Facility manager data records (QEST) hold data about re-drives in 2 fields.

1. QESTRSEC - number of IXLLSTE redrives
2. QESTRMEC - number of IXLLSTM redrives

When IBM MQ attempts to get data from the coupling facility but has not specified a large enough buffer for the requested data, XCF returns the **IXLRSNCODEBADBUFSIZE** return code. As a result, IBM MQ redrives the request to get the data using a larger buffer.

You may also see re-drives if you have many messages on a shared queue with a common value in the index type and subsequently use MQGET whilst specifying the common value. This is discussed in more detail the “[Common MSGID or CORRELID with deep shared queues](#)” section.

Why do I see many re-drives in the statistics report?

For performance reasons, it is better to use a buffer that is close to the size of the data (including headers) to be gotten - as a result, the initial buffer size used by MQ is 4KB and this will be increased or decreased as necessary.

IBM MQ uses a range of buffer sizes i.e. 256 bytes, 512 bytes, 1024 bytes, 2048 bytes and then the size of the buffers increase in 4KB blocks up to 63KB.

IBM MQ stores the size of the last message and uses this size for the buffer on the next get, as typically it is expected that all the messages on the queue are the same size. If the message size increases above the size of the current buffer, a redrive will occur. For example, consider a shared queue has 5 messages of varying size (including MQ headers).

Message	Size of Message (KB)	
1	5	Redrive occurs
2	6	
3	8	
4	10	Redrive occurs
5	1	

In this example, you would see 2 re-drives – associated with message 1 (as it is larger than the initial 4KB buffer size) and message 4 (as it is larger than the next boundary (8KB)).

In version 6.0.0, MQ may resize the buffer immediately if the buffer is deemed too large, so consider the following example:

Message	Size of Message (KB)	
1	1	
2	0.5	
3	1	Redrive occurs
4	4	Redrive occurs

NOTE:

- Message 1 does not require a re-drive but the 4KB initial buffer was inefficient, so the buffer is resized to 1KB for subsequent attempts.

- Message 2 can be gotten into a 1KB buffer, so is also successful without a re-drive, but the buffer is inefficient, so the buffer is resized to 512 bytes for subsequent attempts.
- Message 3 does not fit into a 512 byte buffer, so we get a re-drive and the buffer is set to 1KB for subsequent attempts.
- Message 4 does not fit into the 1KB buffer, so again there is a re-drive and the buffer is set to 4KB.

WebSphere MQ version 7.0 changed the behaviour of the sizing down process – it only sizes the buffer down if the buffer was too large for 10 consecutive messages. In the previous example, there would have been no redrives.

Effect of re-drives on performance

If there are a high number of re-drives, it would indicate that the messages being retrieved by MQGET are of varying size – it may be that the messages are not varying in size by much but they are near to one of the boundaries – e.g. some messages are 4080 bytes and some are 4100 bytes.

When there are a large number of re-drives, the cost per transaction may be higher and the transaction rate lower than when the messages do not cause re-drives.

To show how the cost of re-drives can affect the cost and rate of transactions, the following scenario was run:

- A version 6.0 queue manager in a queue sharing group is started
- 5 requester tasks each put non-persistent messages to a single shared queue and then wait for a reply message on a shared reply-to queue that is indexed by MSGID. Each requester will then repeat until told to stop.
- 4 server tasks that get messages from the shared input queue and MQPUT a reply message to the shared reply-to queue
- Measurements were run on a single LPAR using z/OS 1.9 with 3 dedicated processors on a z10 EC64.
- Message sizes specified exclude headers

There are 3 test cases measured:

1. All messages are 2KB
2. All messages are 4KB
3. 3 in 5 messages are 2KB, 2 in 5 messages are 4KB

Message Size	Transaction Rate / Second	Cost / Transaction (microseconds)	Single Retries on Application Structure
2KB	9758	291	0
4KB	9405	302	1
Mixed	9117	313	457,825

As can be clearly seen from the above measurements, using mixed size messages that cross re-drive boundaries on the queue results in a high re-drive count.

It also shows that the transaction rate and cost per transaction is worse than when using larger fixed sizes messages.

It should be noted that if these measurements were repeated using WebSphere MQ v7.0 or later, it is likely there would be minimal re-drives.

Batch delete of messages with small structures - CFLEVEL(4) and lower

When shared queue messages are gotten destructively, the queue manager saves the pointer (PLEID) to the storage used to hold the message in the coupling facility in a batch delete table. When this table is full, MQ initiates an SRB to request that the data pointed to in the coupling facility is deleted.

There is a batch delete table held in each queue manager for each structure that the queue manager is attached to. Each table can store 341 entries before the batch delete process will be initiated.

This means that when putting and getting messages to an application structure, there may be up to 340 messages in the structure per queue manager in the QSG that are waiting to be deleted.

If the messages are 63KB, this means that there could be 340 messages – or 21.25MB of messages waiting to be deleted – per queue manager.

This means that if there were 30 queue managers in a QSG and messages of 63KB were being used, there could potentially be (30 x 21.25MB) 637.5MB of messages waiting to be deleted.

If the application structure is not large enough to hold these “dead” messages as well as any “live” messages, an application putting to a queue on the structure may get an MQRC 2192 “Storage Medium Full”.

Constraint relief may be gained by:

- Enabling ALLOWAUTOALT(YES) – This may change the ratio of elements to entries which may give an increased capacity.
- Increasing the size of the structure
- Using CFSTRUCT’s at CFLEVEL(5)

Shared Message Data Sets - CFLEVEL(5)

WebSphere MQ for z/OS version 7.1.0 introduced CFLEVEL(5) to store large messages in shared message data sets (SMDS), instead of DB2 for messages larger than 63KB.

This can offer a reduction in management costs for storing large shared queue messages as well as improving the throughput rates achievable.

In addition, a 3-tiered message size offload threshold is introduced to increase the capacity of the CF.

Tuning SMDS

When running messaging workloads using shared message datasets, there are 2 levels of optimisations that can be achieved by adjusting the DSBUFS and DSBLOCK attributes.

The amount of above bar queue manager storage used by the SMDS buffer is DSBUFS x DSBLOCK. This means that by default, 100 x 256KB (25MB) is used for each CFLEVEL(5) structure in the queue manager.

DSBUFS

The **DSBUFS** attribute specifies the number of buffers, taken from above bar storage, that is used to hold a cached copy of the messages. This enables faster access when reading the message from the queue, when performed by the putting (local) queue manager.

Level 1 optimisation: Avoid put time I/O waits

If there are insufficient buffers to handle the maximum concurrent number of I/O requests, then requests will have to wait for buffers, causing a significant performance impact. This can be seen in the CSQE285I message (issued as a response to the “DISPLAY USAGE TYPE(SMDS)” command) when the “lowest free” is zero or negative and the “wait rate” is non-zero. If the “lowest free” is negative, increasing the DSBUFS parameter by that number of buffers should avoid waits in similar situations.

When the message data exceeds one SMDS block, a request to start overlapping I/O operations to transfer multiple blocks concurrently is initiated but this is limited to a maximum of 10 active buffers per request.

If the message is 100KB and DSBLOCK(8K) is set, each message would require 13 buffers but would only be able to use 10 buffers. In this example, it would be more appropriate to use a larger DSBLOCK size to ensure that the I/O operations were completed in an optimum manner, for example if the message were 100KB and DSBLOCK(64K) is set, each message would use only 2 buffers.

Level 2 optimisation: Cache of recently put messages

Once there are sufficient buffers to avoid waits, the next level of optimisation occurs when enough data can be buffered so that when recently written data is read back from the same queue manager, it is possible to find the data in a buffer rather than having to read it from disk. This can save significant elapsed time in the reading transaction.

The following example shows the benefits of tuning the DSBUFS attribute:

- A single queue manager in a QSG with a single CFLEVEL(5) application structure, with DSBLOCK(64K)
- All messages are offloaded to the SMDS dataset.
- A request/reply workload is run.
- There are 12 requester tasks which each put a 63KB message to a single shared queue.
- These messages are got by 1 of 4 server tasks that get and put a reply message in-syncpoint to a second shared queue which is indexed by CORRELLID.
- These reply messages are got by the requester tasks using CORRELLID.
- This is repeated multiple times until the requester tasks are stopped.
- Command DIS USAGE TYPE(SMDS) is used to determine the status of the SMDS buffer.
- The DSBUFS value is altered using ALT SMDS(*) CFSTRUCT(APPLICATION1) DSBUFS(value).
- The test is repeated using DSBUFS values ranging from 1 to 32.

The following 2 charts show the two levels of optimisation. There is a distinct increase in transaction rate when there are sufficient buffers such that the tasks are not waiting for a buffer, i.e. when “wait rate” is 0%. In these examples, a significant increase in throughput is seen (from 600 to 4200 transactions per second).

The second increase occurs when the queue manager is able to get the message data from the buffer rather than having to perform disk I/O operations. In this example, an additional 24% increase in transaction rate is seen (from 4200 to 5300 transactions per second)

Chart: Effect of DSBUFS on transaction rate

Transaction Rate for 63B messages with DSBLOCK(256K)

Varying DSBUFS, 12 Requesters, 4 Servers

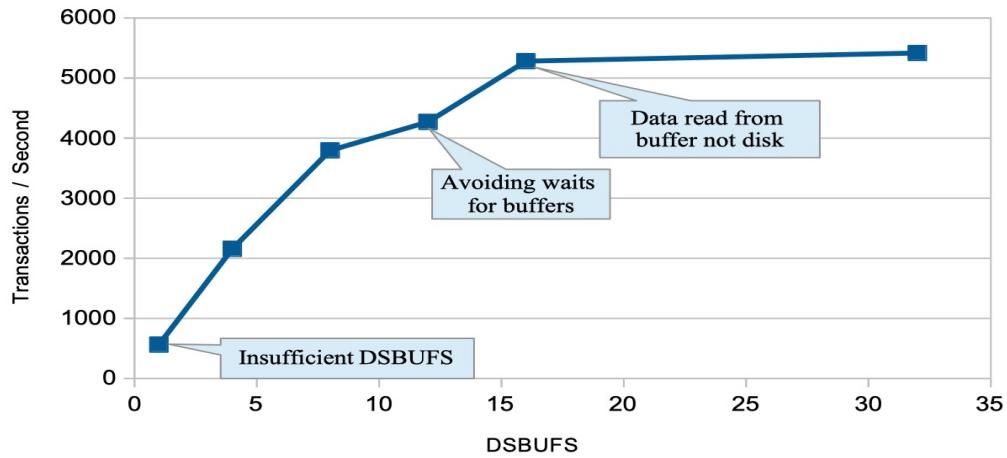
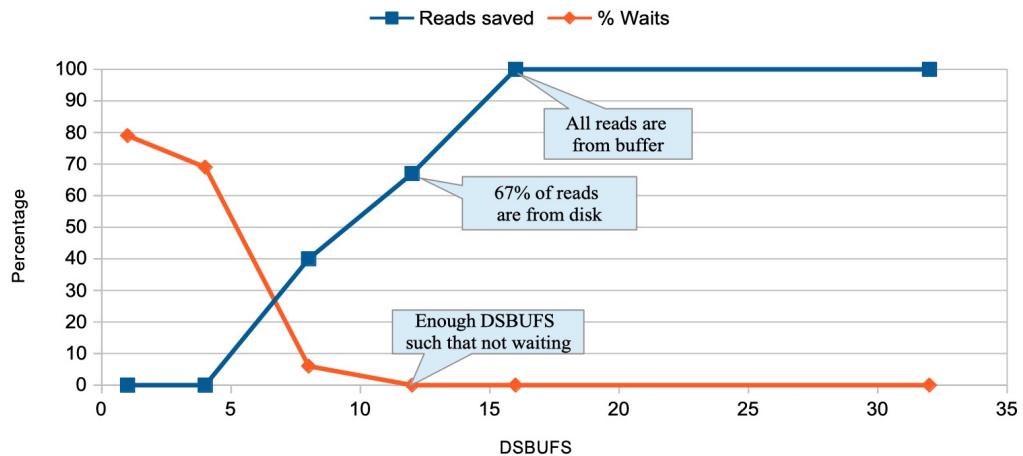


Chart: Using output from CSQE285I message to determine best DSBUFS value

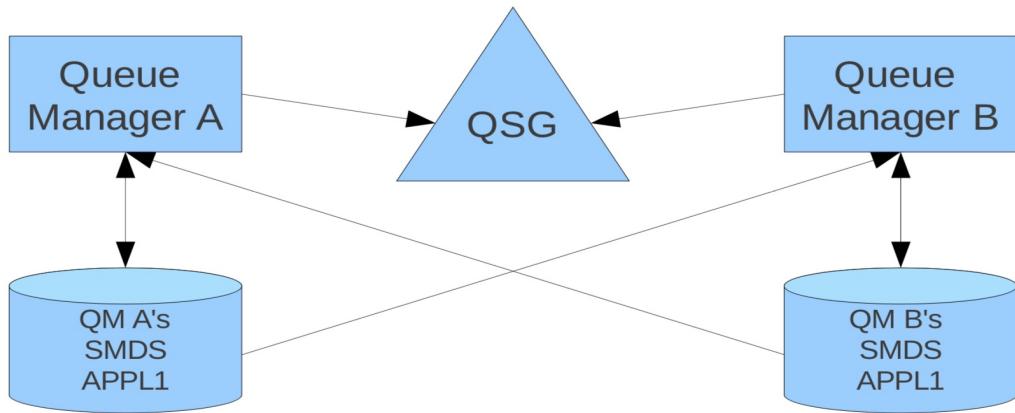
63KB Messages with DSBLOCK(256KB)

Vary DSBUFS, 12 Requesters, 4 Servers



The value of DSBUFS is evident in the preceding charts, however this benefit is only realised when the message is being got from the same queue manager that the message was put on, i.e. the get is from the local queue manager.

Consider the following configuration:



When messages are put to a shared queue by an application connected to queue manager A, the queue manager updates the CF with key information and writes the message to its SMDS. The message will be kept in queue manager A's DSBUFS buffers, as well as the CF and SMDS, until more messages are put to queues in that structure on that queue manager.

Once the buffers are full, they become re-used, so older messages are only stored in the SMDS and CF.

Once the entire message is lost from the buffers and held only in the SMDS, the rate at which the message can be retrieved is comparable, whether the get is from queue manager A (a local get) or from queue manager B (a remote get).

NOTE: Transaction cost is comparable too, however when the get is from a remote queue manager, the queue manager that held the message in its SMDS will perform the delete of the message from the SMDS. This incurs a minor cost to that local queue manager.

DSBLOCK

DSBLOCK is the logical block size in kilobytes in which the SMDS space is allocated for individual queues.

The DSBLOCK attribute divides the SMDS and buffer into blocks for holding the MQ messages.

- By selecting the DSBLOCK size that is larger than the message, storage usage in the buffer is less efficient.
- By selecting a DSBLOCK size that is smaller than the message means that multiple I/O requests are required to store the message on SMDS.

To aid performance, SMDS attempts to overlap I/O requests, but each task is limited to 10 buffers OR where the total size of the buffer is less than or equal to 4MB.

So, maximum overlapped I/O requests is currently 10, but if DSBLOCK(512K) is set, the maximum overlapped I/O requests is 8 (4MB / 512KB).

Understanding how messages are stored in SMDS

Each message is written, starting at the next page within the current block and is allocated further blocks are needed.

Using the default setting of DSBLOCK(256K), each logical block will be 256K (64 pages).

Consider messages that use 100KB:

- Message 1 will be written to block 1 (page 0, using 25 pages).

- Message 2 will be written to block 1 (page 25, using 25 pages).
- Message 3 will cause 2 I/O requests because part of the message will be in block 1 and the remainder will be in block 2.

A larger DSBLOCK decreases space management overheads and reduces I/O for very large messages but increases buffer space requirements and disk space requirements for small messages.

How should I size my DSBLOCKS?

The combination of DSBLOCK and DSBUFS affects how much above bar storage is used by SMDS.

The following lists the order, in preference, for performance sizing the DSBLOCK attribute:

1. Message fits into a single DSBLOCK (i.e. a single I/O request).
2. Message fits into maximum overlapped I/O requests, so that all writes can be requested concurrently.
3. Message is too large and is split into separate I/O requests which cannot be overlapped.

An example of how these configurations affect transaction rate and cost is shown below.

- A single queue manager in a QSG with 1 structure at CFLEVEL(5).
- The SMDS under test has DSBUFS(200) - which has been determined to be sufficient buffers for this work.
- Request/Reply workload is run against a pair of queues using 100KB non-persistent messages.
- The 12 requesters put a message to a request queue and wait for a reply message with a known CORRELID on the reply queue. All requester work is performed out-of-syncpoint.
- The 4 server applications get-with-wait on the request queue and put a corresponding reply message. These gets and puts are performed in syncpoint.
- Measurements are performed on a single z/OS v2r3 LPAR with 3 dedicated processors on z14 (**3906-799**). CF is internal with 4 dedicated processors.
- DSBLOCK is set to the following 3 values:
 - 8K - Message does not fit into 10 buffers, so will require multiple overlapped I/O requests.
 - 16K - Message fits into 7 buffers, so will be written to disk in 1 set of overlapped I/O requests.
 - 128K - Message fits into a single buffer. One I/O request would typically be required to write to disk, unless message spills over from a previously part-used DSBLOCK.

The following table shows the achieved transaction rate and cost per transaction for these tests.

DSBLOCK	8K	16K	128K
Transaction rate/second	1103	2713	3439
Cost / Transaction (CPU microseconds)	544	377	240
RMF “Channel path report” % busy (average for 4 channel paths in use)	34.2	47.34	31.5
Maximum used blocks (CSQE280I)	151	51	7

What does this tell us?

It is more space efficient to use 8K buffers to store 128K messages than either 16K or 128K. However the system allows a higher transaction rate with the larger DSBLOCK size and at less load to the I/O subsystem.

CFLEVEL(5) and small messages

Using the offload thresholds and sizes it is possible to specify that all messages are offloaded, which will increase the capacity of the Coupling Facility.

Because of the way the message header is stored in the Coupling Facility, there is space in the required elements for a message of 122 bytes or less to be stored. MQ is optimised to store these small messages in the CF structure as it does not impact the CF. This means that messages of 122 bytes or less are not offloaded into the shared message data sets.

Who pays for messages stored on Shared Message Data Sets?

Large shared queue messages offloaded to shared message data sets show different cost characteristics compared to large shared queue messages that are offloaded to Db2.

Consider an application that performs an MQPUT and MQGET of a 100KB message to a shared queue, where the message payload is stored in Db2:

- The costs are accumulated to the applications' address space until the queue manager needs to insert the binary large object (or blob) to the Db2 table. At this point the queue manager looks at the Db2 blob threads (as specified in the CSQ6SYSP macro under parameter QSGDATA) to choose and / or wait for an available thread. The queue manager then performs a task switch to that chosen thread and waits for the thread to complete before resuming the applications' thread.
- The select blob thread performs the SQL INSERT to the Db2 table and the cost of the work performed by this thread is attributed to the queue managers' address space.
- With regards to the MQGET, the application is again charged for the work until the queue manager performs a task switch to one of the blob threads to perform the SQL SELECT of the data from the table. If the data is available in the Db2 buffer, the cost of this task may be relatively small.

Compare this to an application that performs an MQPUT and MQGET of a 100KB message to a shared queue, where the message is stored in shared message data sets:

- When a message is put to a queue defined in a CFLEVEL(5) OFFLOAD(SMDS) structure, the queue manager does not need to perform a task switch when writing the message to the shared message data set.
- Similarly when getting a message from a queue defined on a shared message data set, no task switch is required.
- This lack of task switching performed when accessing a message stored on shared message data sets means that:
 - The cost of the put and get is attributed to the application.
 - There is no need to wait for a Db2 blob thread in the queue manager address space.

As a result the application may see an increased cost, but this should be more than offset by a decreasing cost in the queue manager.

There is still some queue manager cost associated with messaging that is not affected by using CFLEVEL(5) OFFLOAD(SMDS), in particular (but not limited to):

- Issuing MQCMIT - this causes the queue manager to task switch to an SRB to ensure the commit is completed.
- Persistent messages - the logging of persistent messages is performed by a single queue manager task.

Db2

IBM MQ uses Db2 with shared queues for a number of reasons, including storing information about queue managers configuration, group objects, channel status and large (greater than 63KB) messages.

The Db2 LOB tablespace supplied definitions specify 32KB buffer pool usage. The Db2 table space requirement is therefore of order:

1. Number of 32KB's required for typical message (including all headers) * maximum number of messages
2. For example, ten thousand shared queue messages of size typically 100KB would require 4 such 32KB's per message and therefore of order 1280MB LOB table space.

There is no queue manager requirement for a particular LOB table space buffer pool size. Other sizes may be used although no performance advantage has been observed using different sizes.

The Db2 supplied definitions specify NUMPARTS 4 to improve maximum throughput. This number can be changed to suit particular workload requirements. We did not find significant benefit from using a greater number of partitions. Each message is stored within a single partition. The partitioning is pseudo-random based on the time the message is MQPUT.

If the partition to which a message is to be MQPUT is full then a 2192 (media full) reason code will be returned even if there is still space in other partitions. Thus it is sensible to add another 10% to LOB tablespace requirements to allow for any uneven usage of partitions. For example, the above calculation of 1280MB split across 4 partitions could sensibly be spread across four partitions each of $(1280/4 + 10\%)$ 352MB.

Db2 universal table space support

MQ version 9.1 provides sample jobs for defining the Db2 tablespaces, tables and indexes. Two sets of samples are provided:

- One for compatibility with earlier versions of IBM MQ, although support for traditional configurations was deprecated in MQ version 9.1.
- One for use with Db2 v12 or later, which exploit Universal Table Spaces (UTS)

Universal Table Space (UTS) support is not specifically a performance feature in terms of MQ's use of Db2, but with Db2 v12 announcing that partitioned non-UTS table spaces are being deprecated, such that they are supported but may be removed in the future, it was necessary to provide the support for when the user is ready to implement UTS.

It should be noted that there is not a direct migration path from the traditional Db2 configuration to Db2 universal table space support, with the suggested path of using the UTS samples when moving to later DB2 versions.

In terms of MQ performance, the preferred option for large shared queue message support is still via Shared Message Data Sets (SMDS), but if there are reasons for using Db2 for large shared message support, then UTS may offer some performance benefits for LOB usage over traditional configurations.

To further complicate performance considerations, the supplied samples for the LOB table spaces use the option “GBPCACHE SYSTEM”. This is optimised for configuration where messages are potentially processed anywhere in the sysplex.

“GBPCACHE SYSTEM” means that at commit after the insert, the changed LOB page is written to DASD instead of the group buffer pool (CF). Only changed system pages (e.g. space map pages) are written to the group buffer pool (CF). When another member comes to delete the message, it will need to read the page from DASD. In those configurations where data sharing is highly prevalent the “GBPCACHE CHANGED” option may be a more suitable option. For more information on this option, please refer to the Db2 Knowledge Center section “[How the GBPCACHE option affects write operations](#)”.

Is Db2 tuning important?

Yes, because Db2 is used as a shared repository for both definitional data and shared channel status information. In particular BUFFERPOOL and GROUPBUFFERPOOL sizes need to be sufficiently large to avoid unnecessary I/O to Db2 data and indexes at such times as queue open and close and channel start and stop.

The Db2 RUNSTATS utility should be run after significant QSGDISP(SHARED) or QSGDISP(GROUP) definitional activity, for instance, when first moving into production. The plans should then be re-bound using SCSQPROC(CSQ45BPL). This will enable Db2 to optimize the SQL calls made on it by the queue manager.

1. For shared queue messages > 63KB
2. Isolate the IBM MQ used Db2 LOB table space into a 32K buffer pool of its own.
3. A group buffer pool definition corresponding to the chosen buffer pool will need to be defined.
We used a group buffer pool CF structure definition with POLICY SIZE: 10240 K.

Why does IBM MQ produce more Db2 rollbacks than I expect?

When running a Db2 log print you may see an unexpectedly high number of rollbacks for the IBM MQ plans that are prefixed CSQ5. The reason for these unexpected rollbacks is that the occurrence of any non-zero return code (including +100 – “not found”) at the end of the Db2 activity will result in a rollback being issued. In particular there may be a high number of CSQ5L prefixed plans that have rollbacks associated with them. The CSQ5L prefixed plan is used when IBM MQ periodically checks for new objects being created.

Shared queue messages > 63KB

Throughput and response time for shared queue messages <=63KB has been and remains most dependent on:

- For persistent messages, I/O rate and total data rate achievable by individual IBM MQ logs in the queue sharing group for all persistent messages processed by individual queue managers.
- For all messages, CF link type and CF power

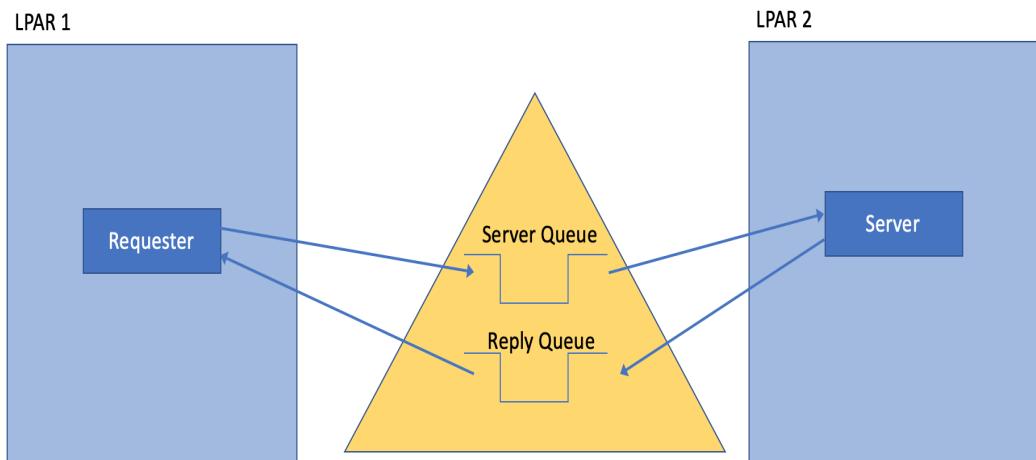
Throughput and response time for all shared queue messages > 63KB is additionally dependent on:

- I/O rate and total data rate achievable by Db2 LOB table DASD.
- I/O rate and total data rate achievable by individual Db2 logs across the data sharing group.
- The percentage of cross Db2 data sharing, which also impacts CPU costs.

- Persistent messages are logged to the IBM MQ log. Db2 LOB table control information is logged to the Db2 log for persistent and non persistent messages. LOB data (the message) is not logged by Db2.
- The queue manager issues all Db2 calls on behalf of any application using >63KB shared queue messages. For virtual storage reasons the messages are segmented, if necessary, into multiple 512KB LOB table entries containing 511KB of message data. This has the following effects:
 - CPU costs for shared queue messages > 63KB are mostly incurred by the queue manager and Db2 address spaces rather than the application.
 - Maximum throughput and response time can be impacted across 511KB boundaries.
 - CPU costs are increased across 511KB boundaries.

100% data sharing is most expensive, that is, where the putter and getter are always in different z/OS's. An example of 100% data sharing is where a shared queue has replaced a channel between queue managers.

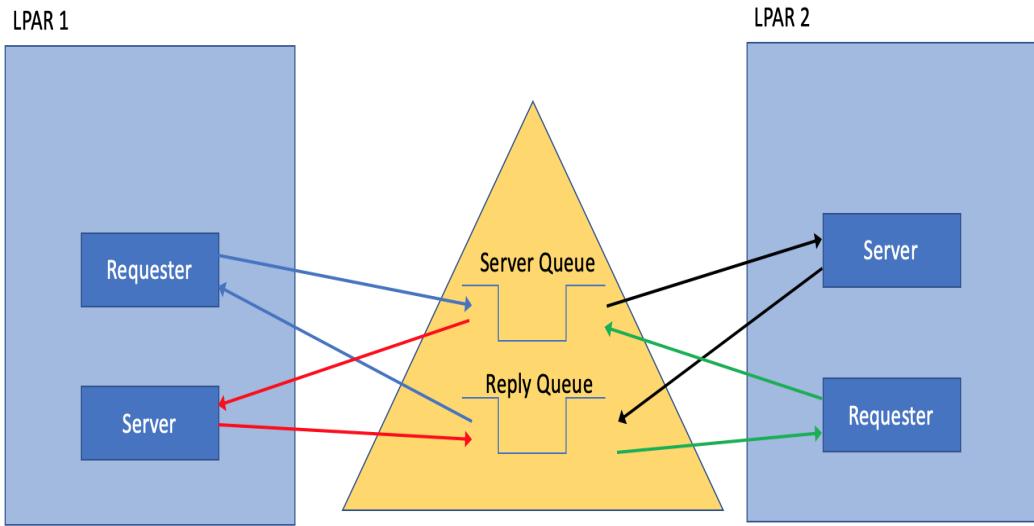
100% data sharing



Requester: MQPUT to server queue, MQCMIT, MQGET-specific-wait from reply queue, MQCMIT
 Server : MQGET-wait on server queue, MQPUT to reply queue, MQCMIT

Where there is effectively a set of cloned systems in a sysplex then data sharing is reduced. For example, consider a set of IBM MQ applications which do all the puts and gets to a set of shared queues. If this set of applications runs on each of 2 queue managers that are connected to separate Db2's in a data-sharing group then there is theoretically 50% data sharing. That is there is a 50% chance that the getter of any particular message will be using a different Db2 to the putter. As more possible destinations are added, the likelihood of the server using a different Db2 to the putter increases, e.g. 66.6% for 3 clones. In reality with cloned systems it is very difficult to ensure an even distribution of workload across the clones.

50% data sharing – Cloned systems



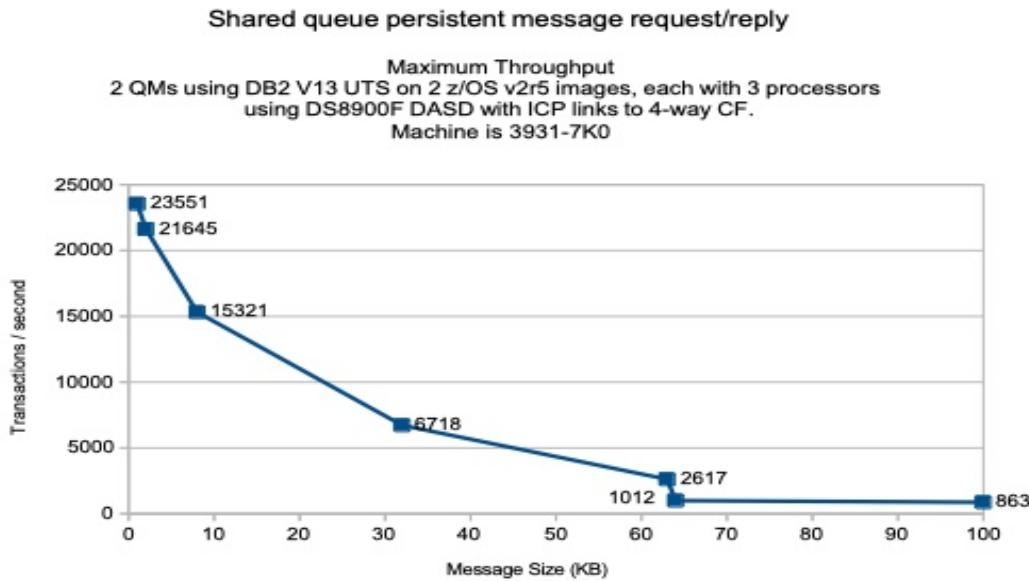
There is also the case where no actual data sharing occurs. This could be because the shared queues requiring messages > 63KB are used only by queue manager(s) each connected to the same Db2. This is most likely to occur in test system setups. Db2 only starts using data sharing locks and group buffers on first LOB table usage by a second Db2.

Finally, out-of-syncpoint non-persistent messages can be MQPUT directly to a waiting out-of-syncpoint MQGETter for shared queues where the waiting getter is connected to the same queue manager as the putter. When this happens there is no interaction with the CF application structure or Db2. Thus the response time and CPU cost is very much reduced when this happens.

NPM SPEED(FAST) receiving channels use out-of-syncpoint MQPUTs for non persistent messages. Thus applications using out-of-syncpoint MQGET on shared queues fed by such channels can benefit. NPM SPEED(FAST) sending channels use MQGET with sync-if-persistent. Thus such channels fed by applications using out-of-syncpoint MQPUTs of non-persistent messages can benefit.

Shared queue persistent message throughput after 63KB transition

The following chart shows the significant discontinuity in throughput at the 63KB transition point. In particular shared queue persistent message throughput drops from 2617 transactions/second to 1012 transactions/second as message size passes 64512 bytes (63KB).



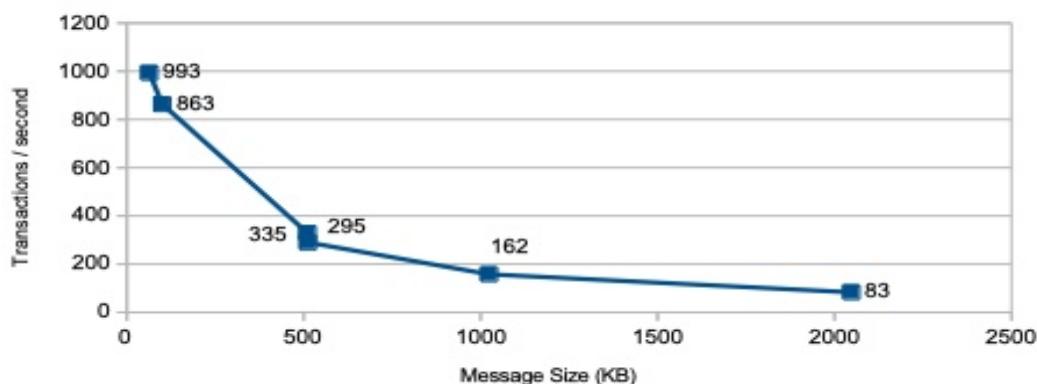
This shared queue scenario is an example of 50% data sharing. There are 2 Db2's with a cloned set of applications using each Db2.

The queue manager issues all Db2 calls on behalf of any application using >63KB shared queue messages. For virtual storage reasons the messages are segmented, if necessary, into multiple 512KB LOB table entries containing 511KB of message data. This has the following effects:

- CPU costs for shared queue messages > 63KB are mostly incurred by the queue manager and Db2 address spaces rather than the application.
- CPU costs are increased across 511KB boundaries – as can be seen in tables following.
- With traditional Db2 configurations, as message size doubles the transaction rate typically halves.
- When Db2 Universal Table Spaces are used, the cost of transitioning to storing MQ message payload in Db2 is reduced, compared to Db2 traditional configurations.

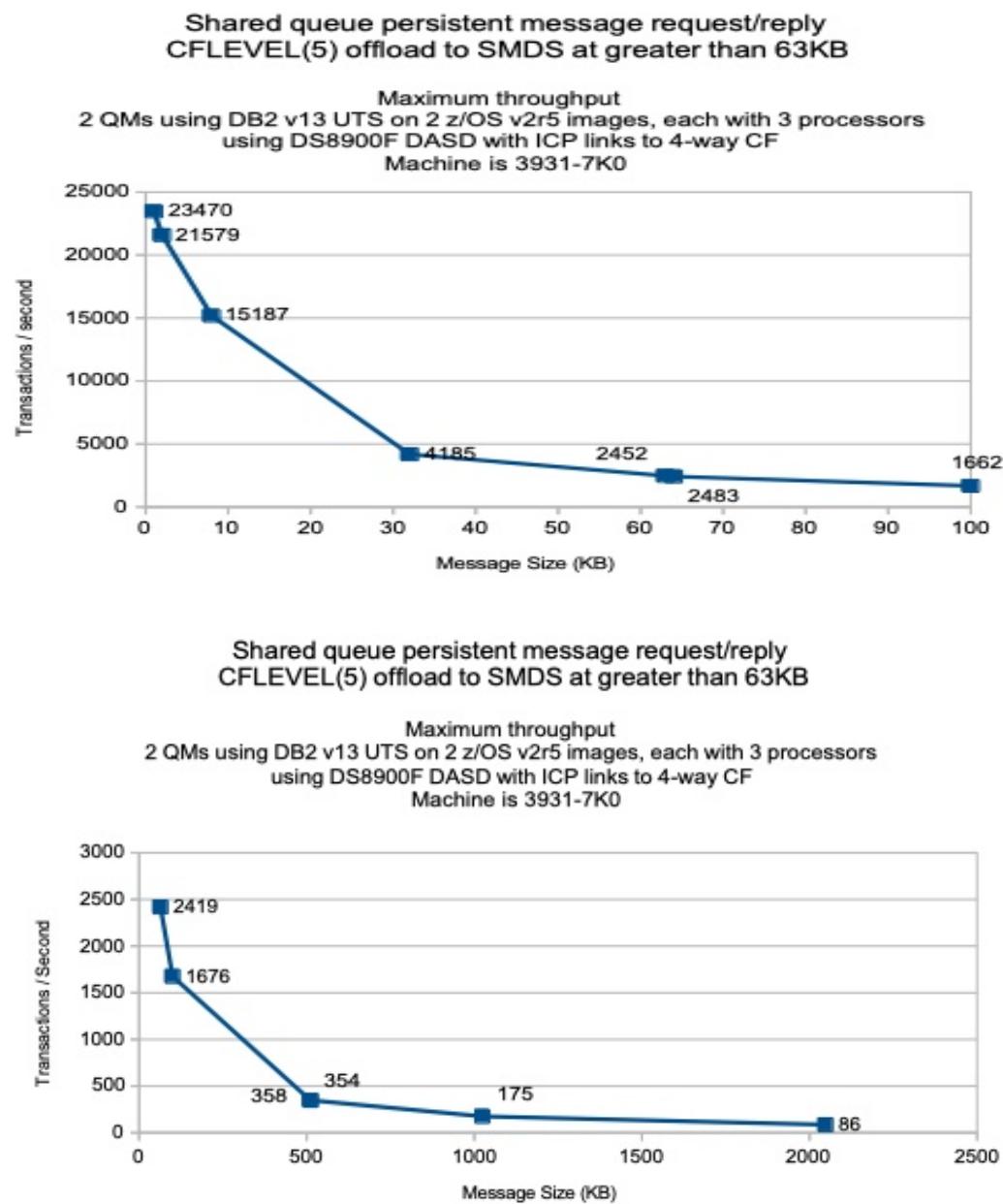
Shared queue persistent message request/reply

Maximum Throughput,
2 QMs using DB2 v13 UTS on 2 z/OS v2r5 images, each with 3 processors
using DS8900F DASD with ICP links to 4-way CF
Machine is 3931-7K0



For comparison purposes, the preceding measurements have been repeated using CFLEVEL(5) using shared message data sets to store the message payload with the default tiering options used. This means that in a coupling facility structure that is not running close to capacity, messages of 63KB or less are stored in the coupling facility.

The following chart shows how using CFLEVEL(5) in conjunction with offloading to shared message data sets with the default offload tiers, the transition from storing the message in the CF to storing the message in the SMDS is significantly smoother than storing the message payload in Db2.



NOTE: As the messages get larger, the benefits of offloading persistent shared queue messages are not obvious from the transaction rate, as the primary factor restricting transaction rate is the queue managers log rate.

Shared queue persistent message request/reply CPU costs

The preceding throughput measurements show the following CPU cost per message (put and gotten) for estimating purposes.

3931-7K0 (partitioned as two 3-way LPARs) CPU microseconds / message		
Message size (Bytes)	CPU microseconds / message using CFLEVEL(4)	CPU microseconds / message using CFLEVEL(5) offload to SMDS
2048	91	91
8192	102	103
32768	149	143
64512	277	174
65536	862	176
102400	1012	210
523264	1618	657
524000	2914	662
1048576	3123	1217

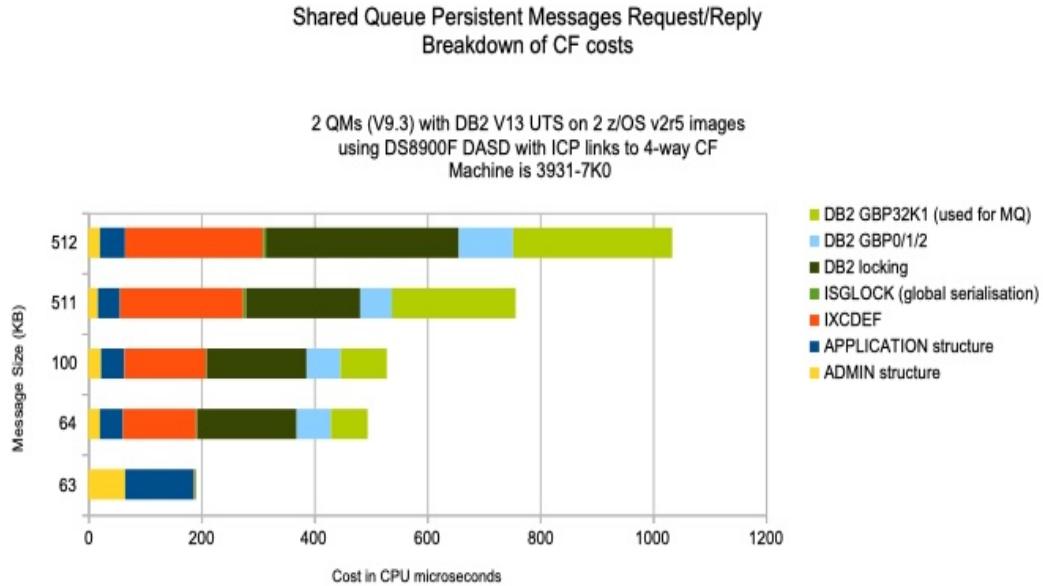
Shared queue persistent message request/reply CF costs

The following table shows the CF CPU cost per message as obtained in the persistent messages in a shared queue environment measurements described previously. The costs are provided for estimating purposes.

3931-7K0 (partitioned as 4-way internal CF) CF microseconds / message		
Message size	CF CPU microseconds / message using CFLEVEL(4)	CPU microseconds / message using CFLEVEL(5) offload to SMDS
2048	24	29
8192	31	25
32768	45	50
64512	96	32
65536	246	31
102400	264	25
523264	331	34
524000	518	35
1048576	739	34

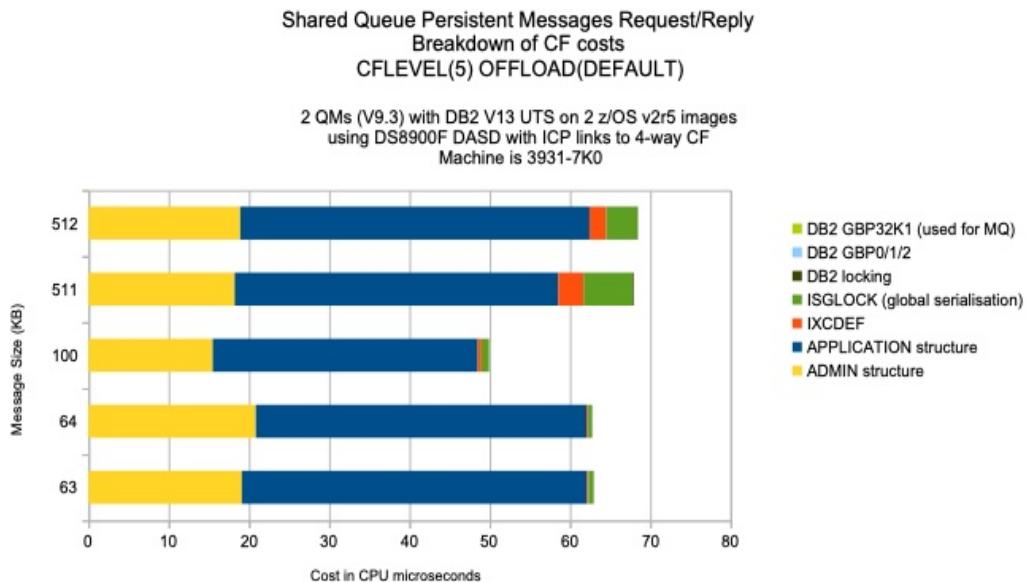
NOTE: The CFLEVEL(5) costs for messages of 32KB and larger are lower in part due to messages being offloaded to SMDS as the offload thresholds were achieved, resulting in the message payload being stored outside of the CF.

The following chart shows the breakdown of cost by component for a variety of shared message sizes.



NOTE: Messages of size less than 63KB see costs incurred by the primarily by the application structure and also the admin structure. As the message size exceeds 63KB, the messages are stored on Db2 tables – resulting in the Db2 lock structures being used more as well as the IXCDEF coupling facility structure that is used to ensure the separate MVS' are synchronized. As the messages reach the size of a Db2 BLOB, the usage of the Db2 buffer pool created specifically for MQ messages increases too.

For comparison purposes, the following chart shows the breakdown of the cost by component for a variety of shared message sizes when offloading messages larger than 63KB to shared message data sets.



NOTE: Using shared message data sets to store messages larger than 63KB is significantly less expensive in CF costs than using Db2 as there is less requirement on the Db2 structures including locking.

Storage Class Memory (SCM)

Storage Class Memory (SCM), also known as CF Flash, is a new feature added on the zEC12 and zBC12 machines, which can be used to provide a high total storage capacity for a CF structure without needing to define excessively large amount of real memory.

On z14, SCM has been moved from Flash Express (SSD on PCIe cards) to Virtual Flash Memory, which allows for simpler management and better performance by eliminating the I/O adapters located in the PCIe drawers.

Example uses of SCM on z/OS:

- Improved paging performance
- Improved dump capture times
- Pageable 1MB memory objects
- Coupling facility (CF) list structures used by MQ shared queues.

Using SCM with IBM MQ

The use of MQ with SCM is discussed in detail in redbook “**IBM MQ V8 Features and Enhancements**” available “<http://www.redbooks.ibm.com/abstracts/sg248218.html>”, which also suggests possible use cases.

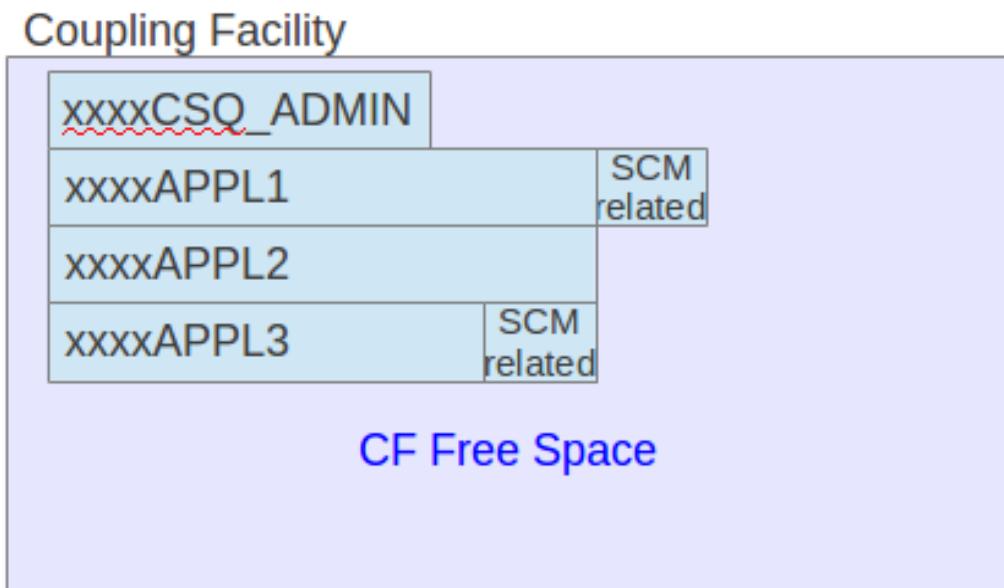
It is suggested that the CF Sizing tool available at [CF Sizer for MQ](#) is used to determine the sizing and impact of SCM on new or existing structures.

Impact of SCM on Coupling Facility capacity

Configuring a structure to be able to use SCM does have an impact on the capacity of the Coupling Facility and existing structure.

- Structure control storage. When a structure is configured to use SCM, the amount of control storage used by that structure increases. This means that a list structure configured with SCM will be able to contain fewer entries and elements than a structure of the same size without SCM configured.
- Augmented storage.
 - Fixed. This storage is allocated when a structure is configured with SCM. This is a small amount and will be allocated even if the structure never uses any SCM.
 - Dynamic. As data from the structure is stored in SCM, augmented space will be allocated from the free real storage in the CF.

Consider the following diagram of a Coupling Facility that has been configured for use by IBM MQ:
In the example, there are 3 application structures defined.



Notes:

- APPL1 and APPL3 have SCM available for additional capacity, and this additional structure control space is denoted by “SCM related”.
- APPL1 and APPL2 have been defined so that they have the same number of entries and elements. This has been achieved by making APPL1 larger to contain the addition SCM related storage.
- APPL2 and APPL3 are the same size but APPL3 has less entries and elements due to SCM related allocations, therefore less capacity in real storage.
- As the SCM storage is used, additional augmented space will be allocated from the CF free space.

How much SCM is available?

The amount of SCM available is ultimately limited by:

- The capacity of CF Flash.
- How much SCM is allocated to the CF.
- How much is actually being used by other structures in the CF.
- How much free space there is in your coupling facility as some may be used for dynamic augmented space. A lack of free space in the CF can limit the amount of space used in SCM.

How do I know I am using SCM and how much?

Using the D XCF,STR,STRNAME=<..> command, which will give output like:

```
..  
SCMMAXSIZE      : 1048576 M  
..  
STORAGE CONFIGURATION          ALLOCATED        MAXIMUM      %  
ACTUAL SIZE:                 4114 M           4114 M    100  
AUGMENTED SPACE:              3 M             77455 M     0
```

This shows that the structure can use up to 1TB of storage and with no data having been written to SCM, there was 3MB of augmented space used from the CF's available free space.

It has been calculated that to use the entire 1TB of available SCM space, 77,455MB of augmented space would be required – therefore the CF needs to have sufficient capacity available.

ALLOWAUTOALT(YES) usage with SCM

When ALLOWAUTOALT(YES) is defined, the threshold for altering the ratios of entries to elements is specified using the FULLTHRESHOLD keyword and is typically reached (80% full as configured by the FULLTHRESHOLD keyword) before the SCM pre-staging algorithm is invoked (90% full).

The altering of the entry to element ratio by the ALLOWAUTOALT process can take time and if the structure usage continues to increase whilst the process is continuing, the SCM pre-staging algorithm may start.

Once the SCM pre-staging algorithm starts, the ALLOWAUTOALT entry to element ratio is frozen until the structure no longer has data residing in SCM. If the auto-altering has not completed or even started, the entry to element ratio used by SCM might be inefficient.

Useful guidelines:

- Never over-commit SCM. If you do, then the applications that are relying on it will not get the behaviour that they expect. For example, MQ applications using shared queues might get unexpected MQRC_STORAGE_MEDIUM_FULL reason codes.
- Be aware of the augmented space usage as this can affect the CF usage. If your CF is constrained for space, you may run out of free space before either the structure or SCM is constrained. This will still result in MQRC_STORAGE_MEDIUM_FULL reason codes.

Impact of SCM on Application performance

The following chart shows the maximum rate in MB per second that we were able to write to SCM and read from SCM for a range of message sizes.

These measurements involved accessing messages in priority order using separate putting and getting applications using a single task.

Message Size	Write to SCM	Read from SCM
2KB	228 MB/sec	177 MB/sec
8KB	463 MB/sec	382 MB/sec
16KB	460 MB/sec	357 MB/sec
62KB	868 MB/sec (seeing 3% of requests delayed)	608 MB/sec

When the MQPUT and MQGET rate is below the peak rates shown in the table, pre-staging and pre-fetching is typically asynchronous and as a result, no significant difference in performance was observed whether messages were stored in CF only or CF and SCM.

Non-Sequential gets from deep shared queue

SCM uses the KEYPRIORITY1 algorithm with MQ shared queues to determine the order that messages are written to SCM (pre-staging) and the order messages are migrated back into the CF (pre-fetching).

Both the pre-staging and pre-fetching are typically performed asynchronously to reduce the chance of the application being blocked whilst synchronous I/O to/from SCM occurs.

Pre-fetching using the KEYPRIORITY1 algorithm works on the assumption that messages will be gotten in MQ message priority order. Multiple messages are pre-fetched, the number dependent upon the message size.

When processing messages out of priority order, the pre-staging and pre-fetching function controlled by the KEYPRIORITY1 algorithm is unable to accurately predict which messages can be pre-staged and which messages need to be pre-fetched next. This can result in significantly more expensive MQPUTs and MQGETs.

Consider the following scenarios:

- There are a number of 2KB messages on a shared queue such that all of the messages remain in CF real storage. These messages are randomly got and put by a single application
- There are a number of 2KB messages on a shared queue that has been configured with SCM such that the majority of messages are stored on SCM. These messages are randomly got and put by a single application.

	MQPUT		MQGET	
	Elapsed (microseconds)	CPU (microseconds)	Elapsed (microseconds)	CPU (microseconds)
All messages in CF real	8	8	19	19
Most messages in SCM (z13)	1700	25	3600	80
Most messages in SCM (z14)	8	8	1210	53

In the scenario where messages are randomly got and put primarily from SCM, the difference between the elapsed and CPU time is due to additional time spent processing the migration of the messages to and from SCM, also known as pre-staging and pre-fetching. In these measurements, the CF cost per message is approximately the difference between the elapsed time minus the CPU time.

This means that this type of workload would result in a significantly lower throughput rate as well as an increase in cost to both the MVS and CF LPARs.

The moving of SCM on z14 from PCIe card to Virtual Flash Memory does reduce the response time when accessing messages that the KEYPRIORITY1 algorithm was unable to correctly predict and pre-fetch into CF memory. Even with this improved performance, it is not advisable to use SCM when messages are gotten in a random sequence.

RMF data

The z/OS RMF Coupling Facility report, requested by “SYSPRPTS(CF)” has been updated in z/OS v2r1 to include an “SCM Structure Summary” report. An example of this is shown below:

SCM STRUCTURE SUMMARY													
TYPE	STRUCTURE NAME	SCM SPACE AUGMENTED		LST ENTRY LST ELEM		--- SCM READ ---			--- SCM WRITE ---			SCM AUX ENABLED	DELAYED FAULTS
		MAX/ ALG	%USED	EST.MAX/ %USED	EST.MAX/ CUR	EST.MAX/ CUR	CNT/BYTE AVG ST/ X'FERRED STD_DEV	CNT/BYTE AVG ST/ X'FERRED STD_DEV	CMD/% ALL	CNT/% ALL			
LIST	PRF1APPLICATION1	KP1	1024G	77455M	613417K	3681M	32339	1583.3	32336	1396.7	0	353138	498
				1.0	0.6	4034K	40343K	33910M	1171.2	33907M	839.9	0.0	

This example shows:

- List structure PRF1APPLICATION1 is defined with up to 1TB of SCM storage using algorithm KEY_PRIORITY1. This algorithm determines which messages are least likely to be got next based initially on the MQ message priority and pre-stages those first to SCM.
- The maximum amount of augmented storage (estimated) required should all of the SCM storage be used is 77,455MB. This means that if the Coupling Facility does not have this much free space, it will be possible to run out of space in the CF before the SCM storage is exhausted. Currently only 0.6% of the maximum augmented space is being used.
- It is estimated that the SCM can support 613,417K entries and 3,681M of elements (a ratio of 1:6) at capacity. Currently there are 4,034K of entries and 40,343K of elements (a ratio of 1:10), which means that SCM (and the CF) will run out of elements before entries if the current message size is continued.
- In this interval there were 32,339 SCM read operations initialised transferring 33,910MB from storage class memory to CF and each read operation takes 1583 microseconds.
- In this interval there were 32,336 SCM write operations initialised transferring 33,907MB from CF to storage class memory and each took 1396 microseconds.
 - We might assume that the total amount of data is relatively flat over the interval as the amount of data read and written is similar, and indeed this is as expected as the data is taken from an interval from the “non-sequential gets from deep shared queue” section.
- The delayed faults count is the number of list item references that were delayed due to a fault condition resulting in required access to storage class memory. In this example the value is particularly high as the KeyPriority1 algorithm was unable to predict which messages would be needed. As a result there are multiple faults per request, as indicated by the %ALL value being greater than 100.
 - If the messages are being put or gotten in priority order and the number of delayed faults is high then KeyPriority1 algorithm’s performance may be impacting your application(s) however there is no system tuning available to improve the performance.

Example use cases for IBM MQ with SCM

The section in redbook “IBM MQ V8 Features and Enhancements” discusses 2 use cases for MQ with SCM:

- Improved performance. This uses SCM to increase the number of messages that can be stored on a shared queue without incurring the performance cost of using SMDS.
- Emergency storage. This uses SMDS with message offloading, in conjunction with SCM to reduce the likelihood of an MQRC_STORAGE_MEDIUM_FULL reason code being returned

to an MQ application during an extended outage.

Capacity – CFLEVEL(4 and less) – no offload available - “Improved Performance”

Using SCM in conjunction with a structure defined at less than CFLEVEL(5) will ensure that when the structure reaches 90% full, messages will be pre-staged to SCM. Provided there is sufficient free space in the Coupling Facility, it will be possible to continue to put messages until either the SCM entry or element values reach 100%.

This means that for a 1TB SCM structure with sufficient CF storage for augmented space and the optimal entry to element ratio, it would be possible to store 16.7 million messages of 62KB in the CF Flash storage.

Capacity – CFLEVEL(5) Offload - “Emergency Storage”

When an MQ structure is defined at CFLEVEL(5) and the offload threshold for a particular message size has been exceeded, MQ will store 1 entry and 2 elements in the CF, which is the MQ implementation header plus approximately 130 bytes of message data, as well as writing the remaining message data to the offload media e.g. SMDS.

Given the maximum capacity of a Shared Message Data Set associated to a queue manager is 16TB and up to 32 queue managers may be connected to the structure, the combined SMDS for a single structure may hold 512TB of messages.

This means that for a 1TB SCM structure with sufficient CF storage for augmented space where all messages are offloaded to SMDS and the optimal entry to element ratio (1:2), it would be possible to store 14,310 million messages of 62KB in a combination of CF, SCM and SMDS storage.

Capacity – CFLEVEL(5) – no offload - “Improved Performance”

In order to configure the structure for *improved performance* when using CFLEVEL(5), it is necessary to set the OFFLDnSZ attributes to 64K e.g.:

OFFLD3SZ(64K) OFFLD3TH(90)

This disables offload threshold 3, so would never offload messages less than 4KB to SMDS. This means that using the default rules only messages of 4KB or larger will be offloaded to SMDS. due to the first 2 offload rules. Messages smaller than 4KB will be written in their entirety to CF (for pre-staging to SCM).

Performance / Scalability

Does the CF Structure attribute “CFLEVEL” affect performance?

From version 7.1.0, IBM MQ supports 5 values for CFLEVEL for an application structure.

1. CFLEVEL(1) – Non-Persistent messages less than 63KB.
2. CFLEVEL(2) – Non-Persistent messages less than 63KB.
3. CFLEVEL(3) – Persistent and non-persistent messages less than 63KB.
4. CFLEVEL(4) – Persistent and non-persistent messages up to 100MB.
5. CFLEVEL(5) – As CFLEVEL(4) but allows tiered offloading when coupling facility fills and offload choice of DB2 or SMDS.

Persistent messages can only be used with shared queues when the CFLEVEL attribute is set to 3 or higher and in conjunction with the RECOVER(YES) attribute.

Using CFLEVEL 3 or higher and RECOVER(YES) means that in the event of the Coupling Facility failing, the CF structures can be recovered from the last backup point to just prior to the point of failure.

Using CFLEVEL 3 or higher with RECOVER(NO) means that in the event of the Coupling Facility failing, the messages will be lost. Since these can only be non-persistent messages, and recovery of these messages is not paramount, it is not an unacceptable occurrence.

Measurements using non-persistent messages both in and out-of-syncpoint have shown no significant degradation on throughput nor increase on cost per transaction when using higher CFLEVEL values over a structure that has been defined with CFLEVEL(2).

Using CFLEVEL(5) offers multiple benefits including increased capacity in the CF for messages less than 63KB and significantly improved performance for messages greater than 63KB than the DB2 alternative.

The impact on MQ requests of the CURDEPTH 0 to 1 transition

The 0 to 1 transition in the current depth of a shared queue can affect the cost and throughput of MQ messages.

When an application opens a shared queue for input, the queue manager registers an interest in the queue. This tells XCF to notify the queue manager when certain events occur with this queue. One of these events is when the depth of the queue goes from zero to one – i.e. a message is available on a previously empty queue. When this happens, XCF signals each queue manager that has previously expressed an interest in the queue that there is now a message available. When the signal occurs an SRB task is run within the queue manager that checks to see if additional work needs to be done. By contrast, when the depth of the queue goes from one to two (or more), XCF does not signal all interested queue managers.

Consider the case of a server application processing messages – when there is a low throughput, there are insufficient messages to keep the server application busy. This means that the queue's depth may frequently change from zero to non-zero. This will result in additional CPU being used when XCF signals the zero to one transition.

For example, if there are 10 requester applications connected to 10 queue managers that put messages and get reply messages from a common reply-to queue, when the server application puts a reply message and the queue depth changes from zero, then each of the 10 queue managers will be signalled. One will process the message and the other nine will be signalled, check the request and do nothing further. If the message rate is high enough to ensure this zero to one transition does not occur, XCF does not signal all queue managers that have registered an interest.

When would I need to use more than one structure?

There is no significant performance benefit to using more than one structure within one CF.

There is an impact if you use structures in multiple CFs when processing messages within syncpoint.

You might want to use different structures for administration and transaction isolation. For example:

- If you need to have more than 512 queues defined, you will need multiple structures
- If you have unrelated applications, you may want them to use different structures in case one application has problems and the structure fills up. If the applications are using the same structure then all the applications using the structure will be impacted. If the applications use different structures, a problem with one structure will not impact applications using other structures.
- Your structures may have different requirements, such as duplexing or recovery.
- You may want to limit the space used in the CF by different applications. You could restrict the number of messages on a queue by using queue maxdepth.
- You may want to have your business critical applications using structures on the fastest Coupling Facility, and non-business critical applications using another Coupling Facility

When do I need to add more Queue Managers to my QSG?

There are several reasons that you may need to add extra queue managers to your QSG:

1. Logging - you are being constrained by the rate at which the logs can be written to disk.
2. Channels - you are unable to start any further channels to support workload.

What is the impact of having Queue Managers active in a QSG but doing no work?

There is no significant impact of running with multiple queue managers in a QSG when only a subset of those are actually doing any work compared to running only the queue managers that are processing any work.

As an example, we compared running workload on 2 queue managers in a QSG when:

- These were the only 2 queue managers active
- When there were 12 queue managers in the QSG, and 10 were “idle”.

The CPU cost per transaction were the same in each case and the throughput dropped by less than 1% when there were 12 queue managers available (the throughput dropped from 1536 to 1526 transactions per second when using 1KB non-persistent messages).

Note: If shared queues have been defined to the IMS Bridge storage class, the “idle” queue managers will process IMS Bridge messages. For further details, see the section titled “[Putting messages from MQ into IMS](#)”.

What is a good configuration for my shared queues?

In this section, many shared queue considerations have been discussed. For the simplest case, we would advise that there are a minimum of 2 application structures:

- One application structure can be used for non-persistent messages and can be configured with CFLEVEL(3)² or the highest supported CFLEVEL as this structure does not need to be

² CF level 3 onwards is specified as in the event of a coupling facility failure, the queue manager will remain available.

backed up. This will reduce recovery times.

- Since these are non-persistent messages (i.e. not critical to the business) is there really any need to duplex this structure? Duplexing a structure will reduce the through-put and increase the cost of the transaction.
- Using CFLEVEL(3) or higher with RECOVER(NO) means that only non-persistent messages up to 63KB can be put to queues defined in this structure.
- If large (i.e. greater than 63KB) non-persistent messages need to be stored in shared queues, the structure will need to be defined with CFLEVEL(4). Again by setting the RECOVER(NO) attribute, only non-persistent messages can be put to queues defined on this application structure.
- If there is a requirement for a large capacity, use CFLEVEL(5) with offload to SMDS as this will allow the system to offload messages to the shared message data sets when the CF usage reaches the tiered offload thresholds.
- If large messages are being used, CFLEVEL(5) offers a significantly less expensive offload option than DB2.
- The other application structure can be used for persistent messages and set to a higher CFLEVEL (or RECOVER(YES) if already using CF level 3 or higher) such that it may be backed up in the event of a failure. If desired it may be duplexed, either to a local secondary CF or to a physically remote coupling facility.
- Do you really need to cross application structures when running in-syncpoint? Bear in mind that there is an associated cost when crossing structures.
- If more than 2 application structures are required, consider the use of them – do they need to be duplexed? Do they need to be backed up in case of a failure of the CF?
- If running with a duplexed application structure it is advisable to have the CSQ_ADMIN structure duplexed too.

Shared queue persistent messages

Throughput for persistent messages in shared queues is ultimately dependent on the MQ log bandwidth of each queue manager in the queue sharing group. Additionally it depends on general shared queue considerations as follows

Shared queue performance affecting factors

- For messages up to 63KB (64512 bytes)
 - z/OS heuristics which can change CF calls from synchronous to asynchronous
 - The type of link(s) between individual z/OS's and the CF(s).
 - This affects the elapsed time to complete CF calls and so influences the heuristics.
 - The CF machine type and CF CPU %BUSY
- For messages larger than 63KB
 - As above for up to 63KB messages plus the throughput performance of the DB2 data sharing group tablespace used to store these messages.

The performance affect of these factors can vary significantly from one machine range to another.

Coupling Facility Resource Management (CFRM) attributes

z/OS v2r3 introduced 2 new CFRM policy attributes, LISTNOTIFYDELAY and KEYRNOTIFYDELAY, to complement the existing SUBNOTIFYDELAY attribute.

In our tests, we found that only KEYRNOTIFYDELAY offered benefits to MQ shared queue workloads.

The CFRM policy attribute KEYRNOTIFYDELAY allows the user to specify in their policy definitions the amount of time between notification of a system-selected keyrange monitoring exploiter instance and the notification of the other exploiter instances. The exploiter instance that receives the initial notification is selected in a round-robin fashion.

KEYRNOTIFYDELAY may be applied to one or more CF structures that queue managers in a queue sharing group connect to. It is important to recognise that this means that the KEYRNOTIFYDELAY will apply to all queues in those structures.

Applying KEYRNOTIFYDELAY will result in the CF selecting a single queue manager to notify that the shared queue has seen a zero to non-zero transition, i.e. messages arriving on the queue.

Note:

- If the selected queue manager empties the queue within the delay time, i.e. returning to zero state, then notification of the other registered queue managers will be bypassed.
- If the queue is not empty when the delay expires, all other queue managers in the QSG that have registered an interest in the queue will be notified.

Subsequent zero to non-zero transitions will result in the CF selecting a different queue manager as the first to be notified.

There are several instances where KEYRNOTIFYDELAY may be beneficial to a workload:

- When the workload is skewed to particular LPARs.
- When there is a low messaging rate with large numbers of server tasks waiting for messages - resulting in many unsuccessful (empty) gets for each successful get.

For more information on this attribute, refer to blog: [z/OS v2r3 new CFRM policy attributes and impact to MQ](#).

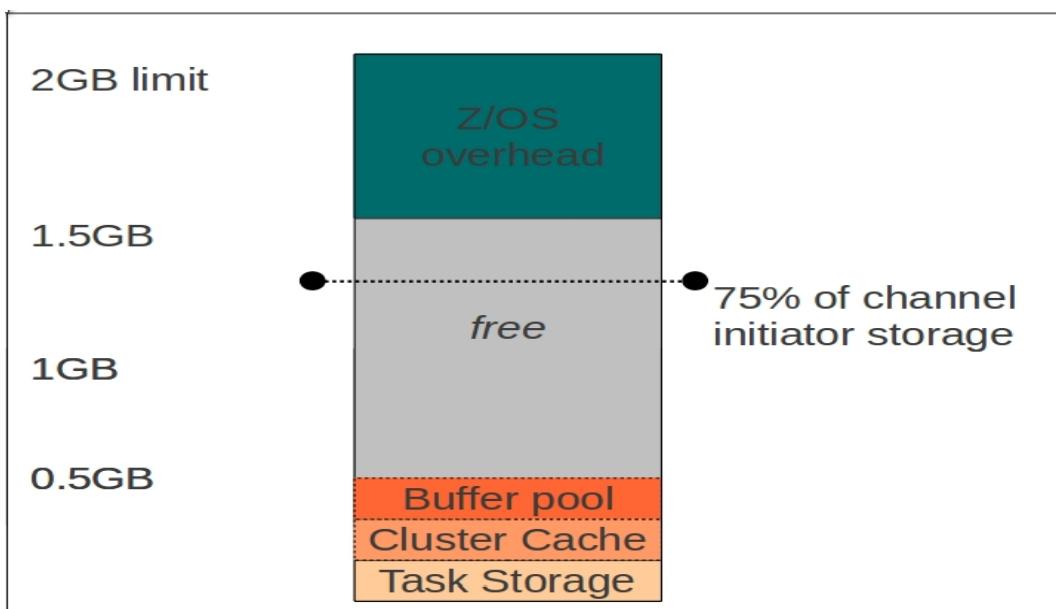
Chapter 3

Channel Initiator

What is the capacity of my channel initiator task?

The channel initiator address space is limited to 2GB of storage and this storage is used for tasks and memory usage. The diagram below shows how the storage within the address space can be used:

Channel initiator storage usage



Notes:

- This above diagram is not an address space map. The coloured blocks indicate approximate relative sizes only. In addition the “z/OS overhead” includes fixed ECSA allocation.
- Task storage consists of adapters, dispatchers, SSL tasks, DNS task and pub/sub tasks.
- The cluster cache stores data about cluster subscriptions. If the queue manager does not have any cluster channels defined, this storage will not be allocated. The storage usage may vary – if the CLCACHE is set to STATIC, 2MB will be used but if CLCACHE is DYNAMIC, the storage usage is 2MB but may grow.
- The buffer pool usage is explained in the following section but does not relate to the queue

manager buffer pool usage and the size may change depending on a number of factors including message sizes.

- Storage usage can be tracked by the CSQX004I message that appears in the channel initiator on an hourly basis unless the storage is used more rapidly.

Version 8.0 introduced channel accounting and statistics, which uses storage from above the 2GB bar, so does not impact the number of channels that the channel initiator is able to support.

Channel initiator task storage usage

In version 9.3, the storage used by each task type can be estimated as:

- Adapter 150KB
- Dispatcher 102KB
- SSL Task 1.45MB

What limits the maximum number of channels?

The maximum number of channels is limited by:

- Channel initiator virtual storage in the extended private region (EPVT) which applies to all channel types including CHLTYPE(SVRCCONN) channels (thin clients)
- Possibly, by achievable channel start (or restart after failure) and stop rates and costs. These increase with the number of channels represented in the SYSTEM.CHANNEL.SYNCQ.

In WebSphere MQ for z/OS version 6.0.0 and earlier, every non-SSL channel uses about 140 KB and every SSL channel about 170KB of extended private region in the channel initiator (CHINIT) address space. Storage is increased if messages larger than 32 KB are being transmitted. This increased storage is freed when either a sending or thin client channel requires less than half the current buffer size for 10 consecutive sends or a heartbeat is sent or received.

WebSphere MQ for z/OS version 7.0.0 introduced the concept of channel initiator buffer pools where a pool of storage is maintained per dispatcher task. The size of the message being processed by the channel initiator directly affects the size of the memory footprint. When the channel initiator determines that there is a shortage of storage available, the channel initiator will attempt to release some of the allocated storage pools.

The upper limit is likely to be around 9000 non-SSL or 9000 SSL channels on many systems as EPVT size is unlikely to exceed 1.6GB.

How many channels can a channel initiator support?

This depends on the size of the messages flowing through the channel.

A channel will hold onto a certain amount of storage for its lifetime. This footprint depends on the size of the messages.

Message Size	1KB	32KB	64KB	4MB
Footprint (KB) per channel (channel initiator)	90	100	109	1125

Overhead of message size increase on 1KB messages		+10	+19	+1035
--	--	-----	-----	-------

If the message size over a channel varies significantly, the effect of channel initiator buffer pools can play a more significant effect.

For example, running 1000 channels with 1KB messages and 50 channels with 64KB messages would use:

- 90KB * 1000
- 109KB * 50
- Which gives a total of 95,450KB, out of the “free storage”

How many SVRCONN channels can a channel initiator support?

The following table shows the footprint of a SVRCONN channel being run. This is shown in KB per channel.

SHARECNV	1KB Messages	32KB Messages	64KB Messages
0	83	161	187
1	156	232	263
10 where each channel instance has 10 shared conversations	245	346	373

NOTE: With 1KB messages, a SVRCONN channel defined with SHARECNV 10 (or greater) will use approximately 24.5KB per shared conversation (CURSHCNV). This means that if a SVRCONN channel is defined with SHARECNV(1000) but has only 10 shared conversations (as reported by DIS CHS(*) CURSHRCNV), the footprint would be 245KB for the channel (or 24.5KB per conversation).

On a system that has an EPVT size of 1.6GB, this means that running server-connection type channels with SHARECNV(0) with 1KB messages, the maximum number of clients that can be connected should be able to reach the IBM MQ defined limit for MAXCHL of 9,999.

For a server-connection channel with a SHARECNV value of 1, it would require 1747MB of storage to start 9,999 channels, so the maximum number of channels would be 9,370.

However, if shared conversations are being used on a server-connection channel with SHARECNV value of 10, it is possible to have 6,930 channels running – with 69,300 conversations.

If the greater memory usage of using a non-zero SHARECNV channel outweighs the benefits such as client heartbeating, read ahead (as allowed using the queue attribute “DEFREADA”) and client asynchronous consume, it is advised to alter the default SVRCONN channel definition to specify SHARECNV(0).

For multi-threaded clients running at version 7.0 or later, e.g. Java applications, define an additional SVRCONN channel with SHARECNV(10) and ensure that these multi-threaded clients use the new channel.

Does SSL make a difference to the number of channels I can run?

Yes. Typically SSL uses an additional 30KB for each channel but for messages larger than 32KB can be higher.

Channel initiator buffer pools

In WebSphere MQ version 7, the concept of channel initiator buffer pools were introduced. These are typically used when the size of the messages flowing across the channels varies significantly.

Each dispatcher has a pool of storage buffers available to handle requests.

In addition, there is a global pool for large storage requests which are shared across all dispatchers.

The dispatcher buffer pools handle requests up to 36KB which allows a full 32KB buffer to be held (which is a common size at the dispatcher level). The dispatcher buffers work with a range of sizes:

- 4KB
- 8KB
- 16KB
- 32KB
- 36KB

The size selected is based upon the required size and the channel initiator searches the available buffers. If no buffer is available of the appropriate size, the next size up is checked. If no buffers are available, a new buffer will be allocated.

The global pool has a set of buffer pools with sizes:

- 64KB
- 128KB
- 256KB
- 512KB
- 1MB
- Larger than 1MB

Each buffer pool, whether a dispatcher or global buffer pool, except the “larger than 1MB” buffer may have up to 50 free buffers in addition to in-use buffers and the usage will depend on how many buffers are actually busy. The “larger than 1MB” buffer pool may only have up to 5 free buffers, so the “larger than 1MB” free buffer pool may vary in size up to 500MB¹, but the “in-use” pool may be as large as the channel initiator can support.

Q+A: What uses a buffer pool buffer?

Typically a SVRCONN channel may have 3 buffers and an MCA channel may have 2 buffers – one for getting / putting the message and a second buffer for sending/receiving the message from the communications protocol.

As previously mentioned a started channel will hold onto storage for its life-time so unless the message sizes significantly change, the buffer pool usage will not vary greatly.

Q+A: What happens when my channel stops?

When a channel stops, the associated buffers are released back to the buffer pools. This can make the storage footprint appear higher than expected as the storage is not released until the channel initiator determines that it has reached 75% utilised and drives its scavenger task. For example, if a channel is started to send 100MB messages, upon ending, the storage will be returned to the global buffer pool and will not be freed immediately. This will mean that the CSQX004I message will still include the 100MB buffers in its calculation even though the 100MB buffers may not be in use at the time.

¹Based on a maximum message size of 100MB

What happens when the channel initiator runs out of storage?

The messages logged by the channel initiator reporting storage usage should mean that storage becoming constrained is not a surprise.

Given the nature of the buffer pools, some variation in storage usage should be expected if a widely varying message size is seen through-out a period of time. It may be that typical day-time workload involves smaller messages and over night, there is a time where a set of large messages are flowed. In this case, it may be observed that the scavenger task is started to release storage that is allocated but not in use.

If a dispatcher attempts to process a message requiring a large buffer and there is insufficient space available and the scavenger hasn't been given time to run, the dispatcher task will fail, resulting in the CSQX112E message being logged, e.g. “[CSQX112E @VTS1 CSQXDISP Dispatcher process error, TCB=xxxxxxxx reason=0C4000-00000004](#)”. Manually restarting the channel should resolve this problem provided sufficient storage is available.

If the scavenger message (CSQX068I) is appearing occasionally, this suggests that the channel initiator is running with its optimum working set of buffer pool sizes which changes when larger messages are flowed.

If the scavenger message is occurring frequently, this suggests that the channel initiator address space does not have sufficient capacity for the workload being run and an additional channel initiator (and associated queue manager) may be required.

Channel Initiator Scavenger Task

As previously mentioned, the channel initiator reports the memory usage on an hourly basis and more often when the storage usage increases significantly.

As part of this storage monitoring, the channel initiator will start a scavenger process if the total memory usage reaches or exceeds 75% of the available storage to release the free buffers from the channel initiator buffer pools.

If the channel initiator only uses long-running channels, the scavenger task may not be started for up to 1 hour after detecting the storage change.

NOTE: The X004I message is logged on an hourly interval or when the storage usage changes by more than 2%. The scavenger task is initiated following the storage usage exceeding 75% and the subsequent logging of the X004I message. If the buffers are still in use when the scavenger tasks is started, it will be unable to free any storage and so will only be able free those buffers when the workload is complete. Typically this is driven by a message on the SYSTEM.CHANNEL.INITQ (e.g. start channel) or a subsequent change in storage usage. If no channel state changes occur, it may be an hour before the channel initiator attempts to free available buffers. This can be circumvented by stopping and restarting a channel.

The scavenger task will release all buffer pool storage marked as free except for 1 buffer of 4KB and 1 buffer of 36KB per dispatcher.

Defining channel initiator - CHINIT parameters

The ALTER QMGR parameters CHIADAPS and CHIDISPS define the number of TCBS used by the channel initiator. CHIADAPS (adapter) TCBs are used to make MQI calls to the queue manager. CHIDISPS (dispatcher) TCBs are used to make calls to the communications network. The ALTER QMGR parameter MAXCHL can influence the distribution of channels over the dispatcher TCBs.

CHIADAPS

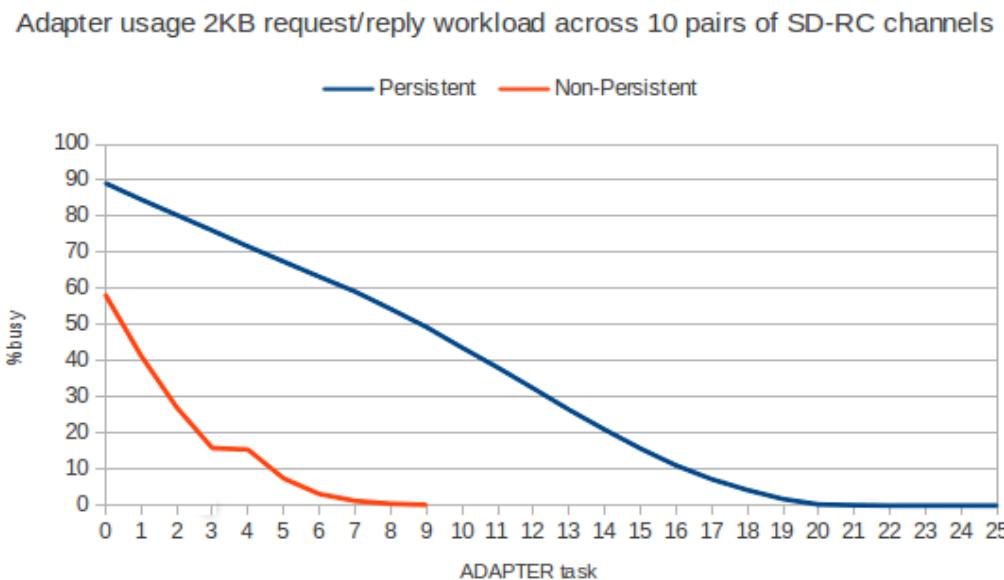
Each MQI call to the queue manager is independent of any other and can be made on any adapter TCB. Calls using persistent messages can take much longer than those for non-persistent because of log I/O. Thus a channel initiator processing a large number of persistent messages across many channels may need more than the default 8 adapter TCBs for optimum performance. This is particularly so where achieved batchsize is small, because end of batch processing also requires log I/O, and where thin client channels are used.

For heavy persistent workloads we recommend CHIADAPS(30) for systems with up to 4 processors and then increase CHIADAPS by 8 for each additional processor up to a maximum of CHIADAPS(120). We have seen no significant disadvantage in having CHIADAPS(120) where this is more adapter TCBs than necessary.

IBM MQ version 8 introduced channel statistics via the use of TRACE(S) CLASS(4). These can be used to determine whether there are sufficient adapter tasks defined to a channel initiator.

The channel initiator uses a pool of adapter tasks and when a request is made, the next available adapter task is used. This results in adapter 0 typically being the most used adapter.

For example the following chart is generated from the adapter reports resulting from workloads using 2KB messages in a request/reply model with 10 outbound and 10 inbound sender-receiver channel pairs where the message persistence is varied.



The preceding chart shows how busy each adapter task is over a 60 second interval.

The non-persistent workload shows that only 8 adapters (0-7) are used and only adapter 0 is more than 50% utilised. For non-persistent messages, the tasks are primarily using CPU.

The persistent workload shows adapters 0 through 9 are greater than 50% busy with the usage tailing off until adapter 18 is less than 5% busy. In this case, the adapter tasks are primarily waiting

for log I/O to complete and whilst this occurs, the tasks are blocked.

A client task that selects messages using message properties can also result in CPU intensive work on an adapter task and there may be benefit in additional adapter tasks in that environment.

Generally, if all the adapter tasks are being used, there may be requests queued waiting for an adapter, so more adapters may offer some benefit to throughput.

CHIDISPS and MAXCHL

Each channel is associated with a particular dispatcher TCB at channel start and remains associated with that TCB until the channel stops. Many channels can share each TCB. MAXCHL is used to spread channels across the available dispatcher TCBs.

The first([MIN\(\(MAXCHL / CHIDISPS \) , 10 \)](#)) channels to start are associated with the first dispatcher TCB and so on until all dispatcher TCBs are in use. The effect of this for small numbers of channels and a large MAXCHL is that channels are NOT evenly distributed across dispatchers.

We suggest setting MAXCHL to the number of channels actually to be used where this is a small **fixed** number.

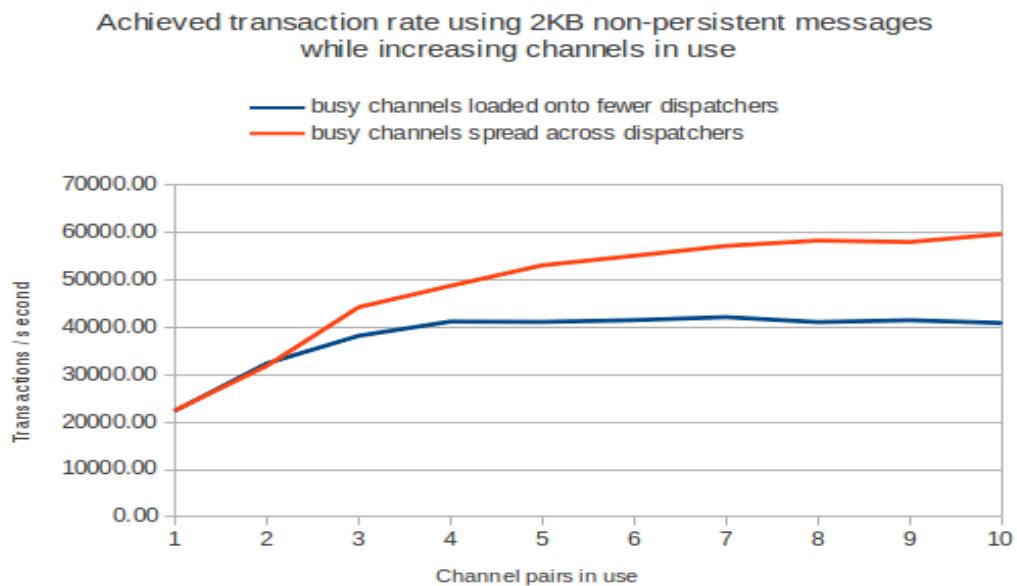
IBM MQ version 8 introduced channel statistics via the use of TRACE(S) CLASS(4). These can be used to determine whether there are sufficient dispatcher tasks defined to a channel initiator.

Performance report [MP1B](#) “Interpreting accounting and statistics data” provides an application, MQSMF, that generates a dispatcher report which can show the usage of each dispatcher. If the report shows dispatchers that have little or no usage co-existing with dispatchers that are showing high usage, the MAXCHL to CHIDISPS ratio may be too high.

For best performance from dispatchers, the system should use as few dispatcher tasks as possible provided they are not being used at capacity. Where larger numbers of channels are being used, we suggest reviewing the dispatcher report but where this is not available, setting CHIDISP(20) where there are more than 1000 channels in use. We have seen no significant disadvantage in having CHIDISPS(20) where this is more dispatcher TCBs than necessary.

The load on each dispatcher can make a difference to the total throughput of the channel initiator. For example the following chart shows the achieved transaction rate with a request/reply workload using 2KB messages as more channels are used. In both cases, there were 3 dispatcher tasks defined.

In these tests, MAXCHL(150) and CHIDISPS(3) were defined.



This means that if all 150 channels were started, there would be 50 channels per dispatcher.

In this test there were 100 channels started but only a subset of channels were actively sending messages as indicated by the x-axis on the chart.

When the busy channels were all on the same dispatcher, the throughput peaked at 40,000 transactions per second even though the dispatcher was 58% busy at its peak.

When the busy channels were shared across all 3 dispatchers, the peak throughput was 50% higher and two of the three dispatchers achieved a higher percent busy than the single dispatcher in the original measurement.

The use of channel compression can affect how many dispatcher tasks are required.

For example the following table shows the report cost of processing a 32KB message that is approximately 40% compressible in the dispatcher task report for MQ running on z15:

Compression type	CPU microseconds per dispatcher request
None	5
ZLIBFAST (hardware compression ²)	12
ZLIBHIGH (software compression)	105

Checking the OMVS Environment

Each channel initiator task uses a number of OMVS threads. Within the channel initiator address space there will be **3³** plus “CHIDISPS” plus “SSL Tasks” OMVS threads.

Use command “**D OMVS,LIMITS**” and review the value of MAXPROCSYS. If the act of adding a new channel initiator or amending an existing one causes the number of MAXPROCSYS to be close to the maximum value, then MAXPROCSYS should be increased.

The SETOMVS command can be used to dynamically change the value, or the value can be updated in the BPXPRMxx parameter values.

²IBM MQ version 8.0 added support for zEDC hardware compression for the ZLIBFAST channel compression option. This is detailed further in performance report [Channel compression on MQ for z/OS](#).

³ This is based on an OMVS thread for CSQXJST, and 2 for CSQXRCTL – one of which is a listener thread for incoming requests directed at the queue manager.

Effect of Changing CHIDISPS

By increasing the CHIDISPS attribute, the number of dispatcher processes is increased. This can be seen in SDSF© using the “PS” option.

If the change is significant or affects a large number of queue managers, these additional processes may cause the system-wide limits set for MAXPROCSYS and MAXPROCUSER to be exceeded. This can be seen when the following messages are logged:

```
BPXI039I SYSTEM LIMIT MAXPROCSYS HAS REACHED 100% OF ITS CURRENT CAPACITY  
BPXI040I PROCESS LIMIT MAXPROCUSER HAS REACHED 100% OF ITS CURRENT CAPACITY
```

Both MAXPROCSYS and MAXPROCUSER are controlled by the BPXPRMxx parameter.

MAXPROCSYS specifies the maximum number of processes that can be active at the same time.

MAXPROCUSER specifies the maximum number of OMVS threads that a single user (i.e. the same UID) can have concurrently active. The channel initiators threads count towards this value, i.e. if there are 10 channel initiators with the same started task userid, and each one has 10 dispatchers, there will be 130 threads for the userid.

When increasing the CHIDISPS value, we recommend amending both the MAXPROCSYS and MAXPROCUSER values by:

```
MAXPROCxxx = MAXPROCxxx + (Increase in CHIDISPS x Queue managers being amended)
```

```
Where xxx is SYS and USER
```

Tuning Channels

There are a number of options that can be applied to channels that can reduce cost or help identify problems in the appropriate environment.

Channel option BATCHHB

This attribute specifies whether batch heartbeats are to be used. These allow the sending channel to verify that the receiving channel is still active just before committing a batch of messages. If the receiving channel is not active the batch can be backed out rather than becoming in-doubt. By backing out the batch, the message remain available for processing so they could, for example, be redirected to another channel.

Channel option BATCHINT

The minimum time in milliseconds that a channel keeps a batch open. The batch is terminated when one of the following conditions is met:

- BATCHSZ messages are sent
- BATCLIM kilobytes are sent
- The transmission queue is empty and BATCHINT is exceeded.

Channel option BATCLIM

The limit in kilobytes of the amount of data that can be sent through a channel before taking a sync point. A sync point is taken after the messages that caused the limit to be reached flows across the channel.

The default is 5000KB, so if large messages are flowing across the channels, a sync point may be taken after only a small number of messages – possibly after every message.

Channel option BATCHSZ

This attribute is the maximum number of messages that can be sent through a channel before taking a sync point.

The maximum batch size used is the lowest of the following values:

- The BATCHSZ of the sending channel.
- The BATCHSZ of the receiving channel.
- On z/OS, three less than the maximum number of uncommitted messages allowed at the sending queue manager (or one if this value is zero or less). On platforms other than z/OS, the maximum number of uncommitted messages allowed at the sending queue manager, or one if this value is zero or less.
- On z/OS, three less than the maximum number of uncommitted messages allowed at the receiving queue manager (or one if this value is zero or less). On platforms other than z/OS, the maximum number of uncommitted messages allowed at the receiving queue manager, or one if this value is zero or less.

Channel option COMPHDR

The header data sent over a channel can be compressed using this value.

The most noticeable effect of compressing the header will be observed with small messages, for example:

- sending a 4MB message and only compressing the header, will reduce the total data sent by approximately 300 bytes.
- Sending a message of 100 bytes with a 480 byte header and compressing the header will result in reducing the data sent by approximately 300 bytes – down by 50%.

Channel option COMPMSG

This is the list of message data compression techniques supported by the channel.

Typically channel compression is used on high-latency networks or low-bandwidth networks as there is a cost associated with compressing and decompressing the message being sent.

As the compression is performed by a channel dispatcher task, this can add extra workload on the dispatcher task and it would be advisable to re-evaluate the optimal setting of channels to dispatchers ratio.

The zEC12 and zBC12 classes of System z hardware introduced hardware compression using zEnterprise Data Compression (zEDC) Express feature on PCIe.

IBM MQ version 8.0 added support for hardware compression via the channel compression option “ZLIBFAST”. Compressing messages using zEDC hardware compression can reduce the message costs by 80% compared to compression performed in software.

For further information on the changes for channel compression enhancements both from using software compression, when zEDC is not available, and hardware compression refer to performance report ‘[Channel compression on MQ for z/OS](#)’.

Most recently with the IBM z15, the compression feature is re-located from PCIe to the processor nest. This can significantly reduce the dispatcher wait time when attempting to compress or inflate message data using compression option “ZLIBFAST”. For further information on the benefits of IBM z15 with MQ, refer to ‘[MQ for z/OS on z15](#)’. In particular review the information in the section titled “Channel compression use of on-chip compression accelerator”.

Prior to the IBM z15, the zEDC compression was only available as an optional feature. If the feature was not available, any MQ channels configured with COMPMSG(ZLIBFAST) would revert to compression using software, which is effectively the same as COMPMSG(ZLIBHIGH).

If moving from hardware that did not have the zEDC compression feature available to an IBM z15 where the feature is available on the processor nest and the MQ messages are compressible, then it may be worth re-assessing whether the channel compression may bring benefits to the MQ channel performance.

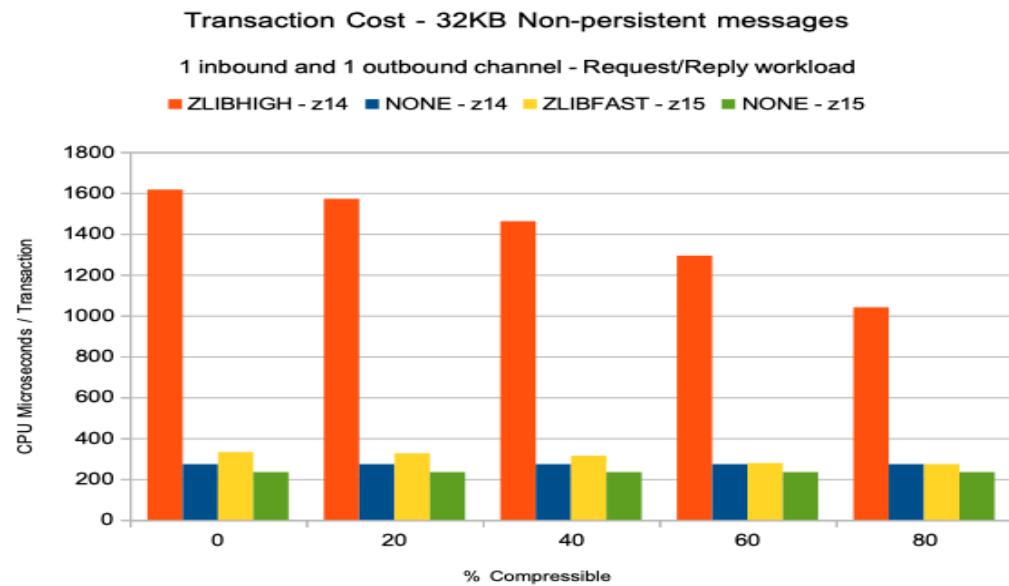
Binary data such as images or video are often not compressible, but data such as XML or fixed format data may be highly compressible. If the data flowing over the MQ channels is text or tagged-data, then compression of 50% or higher may be achievable.

Example: The following scenario demonstrates when a workload on z14 where zEDC compression is not available, is moved to z15 where zEDC compression is available, there may be a reduction in transaction cost as well as an improvement in channel throughput.

In this example, the workload runs a request/reply model using 32KB non-persistent messages between 2 z/OS queue managers. The queue managers are located on separate LPARs on the same physical machine, linked by a dedicated low-latency 10Gb network.

In this request/reply model, the message is compressed and inflated twice - once for the request message and again for the reply message.

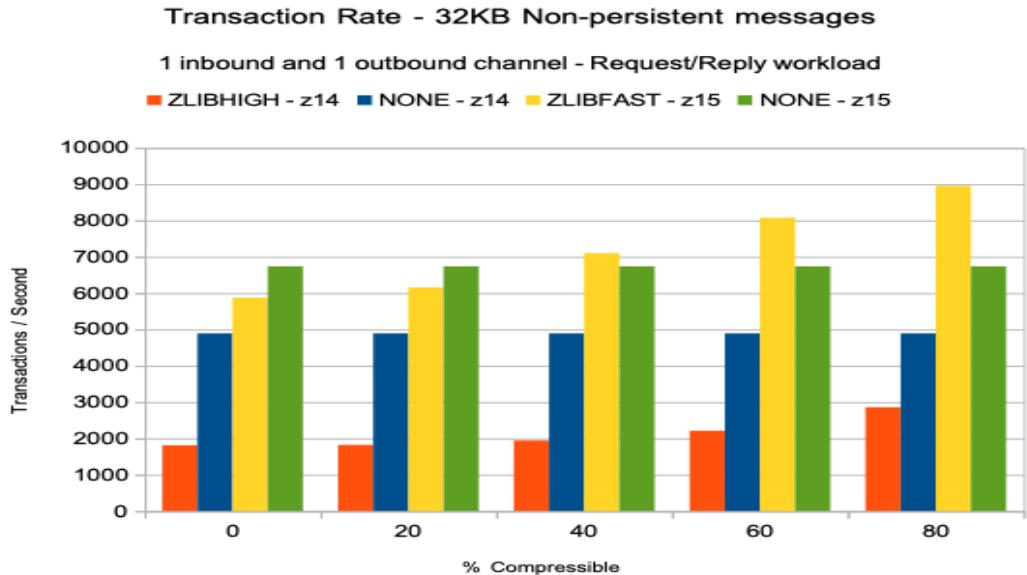
The following chart compares the transaction cost when the message payload increases in compressibility for both z14 and z15.



Notes on chart:

- Compression is not free. The cost of compressing, even on z15 where the majority of the processing is run in the internal zEDC processor, is still higher than not compressing the message data.
- Relying on software for MQ channel compression is expensive - whether directly from ZLIB-HIGH or by falling back due to unavailable compression hardware with ZLIBFAST. The high cost may be offset by other benefits including improved throughput over high latency networks or savings on the frequency of secret key negotiations when using TLS protected channels.
- For more compressible messages, even without TLS protected channels, the transaction cost was lower on z15 with COMPMSG(ZLIBFAST) than the equivalent configuration on z14 with COMPMSG(NONE).

The following chart compares the transaction rate when the message payload increases in compressibility for both z14 and z15.



Notes on chart:

- When moving from z14 to z15 with COMPMSG(NONE), gave a 37% improvement to throughput for this workload.
- Implementing COMPMSG(ZLIBFAST) on z15 gave between 20-83% improvement in throughput over COMPMSG(NONE) on z14, even on a low latency network.
- For this workload, when the 32KB message was 40% compressible, the COMPMSG(ZLIBFAST) measurement exceeded the throughput of the COMPMSG(NONE) measurement.

Channel option DISCINT

The minimum time in seconds the channel waits for a message to arrive on the transmission queue. The waiting period starts after a batch ends. After the end of the waiting period, if there are no more messages, the channel is ended.

A value of 0 causes the message channel agent to wait indefinitely. See the section on channel start/stop rates and costs to see the effect of channels changing state.

Channel option HBINT

This specifies the approximate time between heartbeat flows sent by a message channel agent (MCA). These flows are sent when there are no messages on the transmission queue with the purpose of unblocking the receiving MCA, which is waiting for messages to arrive or for the disconnect interval to expire. When the receiving MCA is unblocked, it can disconnect the channel without waiting for the disconnect interval to expire. Heartbeat flows also free any storage blocks that are allocated for large messages and close any queues that are left open at the receiving end of the channel. To be most useful, the value needs to be less than the DISCINT value.

Channel option KAINST

The value passed to the communications stack for keepalive timing for this channel.

Channel option MONCHL

This attribute controls the collection of online monitoring data for channels. Changes to this parameter take effect only on channels started after the change occurs.

Channel option NPMSPEED

This attribute specifies the speed at which non-persistent messages are to be sent.

- FAST means than non-persistent messages are not transferred within transactions. Messages might be lost if there is a transmission failure or the channel stops whilst the messages are in transit.
- NORMAL means normal delivery for non-persistent messages.

If the value of NPMSPEED differs between the sender and receiver or either one does not support it, NORMAL is used.

SVRCONN channel option SHARECNV

Specifies the maximum number of conversations that can be sharing each TCP/IP channel instance. High SHARECNV limits have the advantage of reducing queue manager thread usage. If many conversations sharing a socket are all busy, there is a possibility of delays. The conversations contend with one another to use the receiving thread. In this situation, a lower SHARECNV value is better.

Tuning channels - BATCHSZ, BATCHINT, and NPMSPEED

To get the best from your system you need to understand the channel attributes BATCHSZ, BATCHINT and NPMSPEED, and the difference between the batch size specified in the BATCHSZ attribute, and the achieved batch size. The following settings give good defaults for several scenarios:

1. For a synchronous request/reply model with a low message rate per channel (tens of messages per second or less), where there might be persistent messages, and a fast response is needed specify BATCHSZ(1) BATCHINT(0) NPMSPEED(FAST).
2. For a synchronous request/reply model with a low message rate per channel (tens of messages per second or less), where there are only non-persistent messages, specify BATCHSZ(50) BATCHINT(10000) NPMSPEED(FAST).
3. For a synchronous request/reply model with a low message rate per channel (tens of messages per second or less), where there might be persistent messages and a short delay of up to 100 milliseconds can be tolerated specify BATCHSZ(50) BATCHINT(100) NPMSPEED(FAST).
4. For bulk transfer of a pre-loaded queue specify BATCHSZ(50) BATCHINT(0) NPMSPEED(FAST).
5. If you have trickle transfer for deferred processing and the messages are typically persistent, specify BATCHSZ(50) BATCHINT(5000) NPMSPEED(FAST).
6. If you are using large messages, over 100000 bytes you should use a smaller batch size such as 10, and if you are processing very large messages such as 1 MB, you should use a BATCHSZ(1).
7. For messages under 5000 bytes, if you can achieve a batch size of 4 messages per batch then the throughput can be twice, and the cost per message half that of a batch size of 1.

If there are **only non-persistent messages** flowing over the NPMSPEED(FAST) channel, there may be benefits in setting both BATCHSZ and BATCHINT to a high value. Since an NPMSPEED(FAST) channel does not transfer messages within transactions, the messages can be transferred and made available to the getting application before the end of batch occurs. Furthermore since there is no transaction, there is no recovery point, so there is no need to complete the end of batch processing on a timely basis.

However, if there is the possibility of persistent messages flowing over the channel, a high BATCHSZ and BATCLIM may result in a long time before that message is committed by the end of batch flow.

How batching is implemented

The text below describes the processing to send one batch of messages:

DO until BATCHSZ messages sent OR (xmitq empty AND BATCHINT expired)

OR BATCLIM kilobytes sent

- Local channel gets a message from the transmission queue
 - If message is non-persistent and channel is NPMSPEED(FAST):
 - Outside of syncpoint
 - Otherwise:
 - Within syncpoint
 - Adds a header to the message and sends using TCP/IP, APPC, etc.
 - Remote channel receives each message and puts it
 - If message is non-persistent and channel is NPMSPEED(FAST):

```

    ■ Outside of syncpoint
    ○ Otherwise:
        ■ Within syncpoint
END

```

COMPLETE channel synchronisation logic.

Thus,

- A batch will contain at most BATCHSZ messages.
- If the transmission queue is emptied before BATCHSZ is reached and the BATCHINT(milliseconds) time has expired since the batch was first started, the batch will be terminated.
- The achieved batch size is the number of messages actually transmitted per batch. Typically for a single channel the achieved batch size will be small, often with just a single message in a batch, unless BATCHINT is used. If the channel is busy or the transmission queue is pre-loaded with messages, then a higher achieved batch size might be achieved.
- Each non-persistent message on an NPMSPEED(FAST) channel is available immediately it is put to a queue at the receiver, it does not have to wait until end-of-batch. Such messages are known as 'fast messages'.
- All other messages only become available at the end-of-batch syncpoint.

NOTES:

- Fast messages can be lost in certain error situations, but never duplicated.
- All other message delivery remains assured once and once only.
- If batch is reaches BATCHSZ then an end-of-batch indicator flows with last message.
- If the batch is terminated because the transmission queue is empty, or the BATCHINT interval expires, then a separate end-of-batch flow is generated.
- Channel synchronisation logic is expensive. It includes log forces where there are persistent messages or NPMSPEED(NORMAL) channels and an end-of-batch acknowledgement flow from the receiver back to the sender. A low achieved batch size results in much higher CPU cost and lower throughput than a high achieved batch size, as these overheads are spread over fewer messages.

Setting NPMSPEED

For non-persistent messages choosing NPMSPEED(FAST) gains efficiency, throughput and response time but messages can be lost (but never duplicated) in certain error situations. Of course non-persistent messages are always lost in any case if a queue manager is normally stopped (or fails) and then restarted. Thus any business process using non-persistent messages must be able to cope with the possibility of lost messages. For persistent messages NPMSPEED has no effect.

If you have applications with both persistent and non-persistent messages which rely on message arrival sequence then you must use NPMSPEED(NORMAL). Otherwise a non-persistent message will become available out of sequence.

NPMSPEED(FAST) is the default and is usually the appropriate choice, but do not forget that the other end of the channel must also support and choose NPMSPEED(FAST) for this choice to be effective.

Determine achieved batch size using MONCHL attribute

The channel attribute “MONCHL” controls the collection of online monitoring data for channels.

By default, a channel is defined with MONCHL(QMGR). This means that monitoring data is collected for a channel based on the setting of the queue managers MONCHL attribute.

Changes to the value of the channel attribute MONCHL only take effect on channels started after the change is applied.

By altering the queue managers MONCHL attribute to one of the following values LOW, MEDIUM or HIGH, the “[DISPLAY CHSTATUS\(channel\) XBATCHSZ](#)” command can be used to display the size of batches achieved over the running channel.

NOTE: Do not use DISPLAY CHSTATUS (*) when you have many channels unless necessary, as this is an expensive command. It might require many hundreds, or thousands, of pages in buffer pool 0. Buffer pool 0 is used extensively by the queue manager itself and thus overall performance can be impacted if pages have to be written to and read from the page data sets as a result of a shortage of free buffers.

Setting BATCHSZ and BATCHINT

Consider the following 3 types of application scenario when choosing BATCHSZ and BATCHINT.

1. **Synchronous Request/Reply**, where a request comes from a remote queue manager, the message is processed by a server application, and a reply sent back to the end user.
 - This usually implies a short response time requirement.
 - Response time requirements often preclude use of a non-zero BATCHINT for channels moving persistent messages.
 - Volumes on each channel might be so small that end-of-batch will occur after nearly every message even at peak loads.
 - For persistent messages, absolute best efficiency might then be achieved with BATCHSZ of 1 as there is then no separate end-of-batch flow from sender to receiver. Savings of up to 20% of the CPU cost of receiving a message and sending a reply message are typical for small messages.
 - If your volumes and response time requirements permit then set BATCHSZ to x and BATCHINT to y where you typically expect x or more messages in y milliseconds and you can afford the up to y milliseconds delay in response time on that channel.
 - Conclusion, for channels moving any persistent messages is:- Use the defaults unless you really know better!
 - Non-persistent messages on an NPMSPEED(FAST) channel are available immediately they are received regardless of BATCHINT or BATCHSZ. So a non-zero BATCHINT is appropriate for any NPMSPEED(FAST) channel carrying ONLY non-persistent messages.
 - For example, if you expect 30 non-persistent messages per second, set BATCHINT to 2000 (2 seconds) then you will almost always achieve a batch size of 50 (assuming BATCHSZ of 50).
 - The CPU cost saving per message moved is likely to be of order 50% versus that for achieved batch size of 1 compared to achieved batch size of 50.
 - If you can guarantee that only non-persistent messages flow over the channel, it may be beneficial to specify a larger value for BATCHSZ and BATCHINT, such as 100000. This will have the effect of minimising any impact from the end of batch flow, which have an impact to throughput on a high-latency network.
 - If you cannot guarantee that only non-persistent messages flow over the channel, you should consider setting BATCHSZ and BATCHINT such that the maximum time the batch remains active is sufficiently small to ensure your SLA's can be attained.
2. **Bulk transfer of a pre-loaded transmission queue.**
 - Usually implies high volumes, a high throughput requirement but a relaxed response time requirement (e.g. many minutes is acceptable). Thus a large BATCHSZ is desirable.
 - The default BATCHSZ of 50 will give relatively high throughput.
 - Higher BATCHSzs can improve throughput, particularly for persistent messages (and non-persistent messages on NPMSPEED(NORMAL) channels). But might be inappropriate for very large messages sizes, where a failure in mid batch could cause significant reprocessing time.

- Do not use `BATCHSZ > 100` even for messages up to 5KB.
- Do use `BATCHSZ = 1` for 1MB or larger messages as anything larger tends to increase the CPU costs, and can have an impact on other applications.
- `BATCHINT` should be left to the default of 0.

3. Trickle transfer for deferred processing

- You want to transfer the messages as they are generated as cheaply and efficiently as possible. These messages are then either stored for processing in a batch window or are processed as they arrive but it is acceptable that several seconds or minutes elapse from the time the messages were first generated.
- If possible wait until a batch size of 50 is completed. This would require that you set `BATCHINT` to xxxx milliseconds, where more than 50 messages are generated within xxxx milliseconds (assuming `BATCHSZ` greater than or equal to 50).
 - If you left `BATCHINT` at 0 then you would probably achieve an average batch size of less than 2 whatever the setting for `BATCHSZ`. In fact, it is typical that nearly all the batches would consist of just 1 message.
 - This would cost significantly more CPU and logging and some more network traffic than 1 batch of 50 messages.
- Or, consider the case where you expect an average of 20 or more messages per minute and you can accept up to 1 minute delay before these messages are processed. Then:
 - If you set `BATCHINT` to 60000 (i.e. 1 minute) then you will achieve a batch size of 20 (on average, provided `BATCHSZ` greater or equal to 20)
 - If you left `BATCHINT` at 0 then you would probably get 20 batches of 1 message each whatever the setting for `BATCHSZ`.
 - 20 batches of 1 message would cost significantly more CPU and logging and some more network traffic than 1 batch of 20 messages.
- However, a very long `BATCHINT` is probably not appropriate as this could mean a very long unit-of-work and consequent elongated recovery time in the event of a failure. You should usually use a `BATCHINT` value of less than 300000 (5 minutes)

Channel Initiator Trace

TRACE(C) costs can be significant when running a high workload.

Comparing high workload over a single pair of Sender-Receiver channels, we saw between a 25% and 55% increase in the cost of the transaction.

On an idle queue manager/channel initiator, we found that TRACE(CHINIT) added 3% to the cost of running the channel initiator.

IBM MQ version 8.0 for z/OS introduced channel initiator accounting and statistics data. Enabling both of the class(4) trace options typically increases channel initiator CPU costs by 1 to 2%.

Why would I use channels with shared conversations?

When it comes to the V7 clients connecting, using a SVRCONN channel with a SHARECNV of greater than 0 allows the channel to run in V7 mode - which gives several benefits:

- Read-ahead - this allows the queue manager to effectively push messages out to the client when they are available - and the MQ client code maintains its own buffer to hold these messages until the client application is ready for them. This can reduce the line turn-around time giving better performance.
- Client Asynchronous Consume - allows the client to register an interest in multiple queues and receive a message when it arrives on any of those queues.
- Heart-beating over a SVRCONN channel - which allows the queue manager to detect when the client has gone away
- By sharing conversations from a multi-threaded client application, it is possible to decrease the footprint on the channel initiator, allowing more clients to connect.
- In V6, a single SVRCONN channel used around 140KB.
- In V7, a SVRCONN channel with SHARECNV(10) uses around 514K – which equates to each conversation using only 51KB - which allows for significantly more client conversations to be running on a single channel initiator.

There are instances where the customer may choose to run a SVRCONN channel with a SHARECNV value less than 10 for example:

- If the clients are performing persistent workload (or a mixture), they may want to run with a lower SHARECNV setting as the single channel instance will use a single adapter task. This adapter task will be blocked whilst any logging of the persistent message occurs, which can then have a direct impact on the other (non-persistent) conversations in this shared channel.
- If the clients are doing a high-volume of workload - A single channel can only flow a certain amount of data over it and this is similar to the total amount of data that can be flowed over a channel with multiple shared conversations. So if the customer was driving 50MB / second through a SVRCONN and moved to a SHARECNV(10) channel, they would not get 500MB/second through that shared channel. As a result they may want to lower the SHARECNV attribute.

There is an increase in **connection** cost when using non-zero SHARECNV SVRCONN channels of approximately 20% when compared to SHARECNV(0) channels. However when running with SHARECNV greater than 1 and it is more likely that at least one of the conversations from the client is already active, meaning the existing channel can be shared, the connection overhead decreases.

With regards to **workload**, in a like for like workload, e.g. not exploiting read-ahead, the costs are typically similar. If the user is able to use long-running connections and asynchronous puts and get, there may be a reduced transaction cost when using a non-zero SHARECNV value.

Performance / Scalability

Channel start/stop rates and costs

The rate and CPU cost at which channels can be started and stopped varies with the number of channels represented in SYSTEM.CHANNEL.SYNCQ.

A channel is represented in SYSTEM.CHANNEL.SYNCQ if it has ever been started. It will remain represented until its definition is deleted. For this reason we recommend that redundant channel definitions be deleted.

While many users do not start and stop channels with any great frequency, there may still be significant sender channel restart activity after a channel initiator failure.

The following table demonstrates the effect of the number of channels represented in SYSTEM.CHANNEL.SYNCQ on a IBM MQ 9.3 Queue Manager.

Channels in .SYNCQ	Sender Channel START		Sender Channel STOP	
	Channels per second	3931-703 CPU millisecs per START	Channels per second	3931-703 CPU millisecs per STOP
1,000	649	0.22	699	0.22
2,000	552	0.27	800	0.25
4,000	428	0.38	630	0.33

NOTE: For the purposes of this measurement, only the sender-side costs are recorded.

TLS channel start costs

Whenever an TLS-enabled channel pair is started a cryptographic handshake is performed which establishes the authenticity of the channel partners and dynamically generates a secret cryptographic encryption key. This cryptographic handshake increases both the CPU consumption and the elapsed time for the channel start.

On our 3931-703 system we have found the additional TLS costs to be somewhat dependent of the SSL/TLS cipherspec used. With 4000 sender-type channels in SYSTEM.CHANNEL.SYNCQ:

TLS	CipherName	3931-703 CPU milliseconds	Channels per second
TLS 1.3	TLS_CHACHA20_POLY1305_SHA256	0.89	168.6
	TLS_AES_256_GCM_SHA384	0.81	177.2
	TLS_AES_128_GCM_SHA256	0.81	177.1
TLS 1.2	TLS_RSA_WITH_AES_256_GCM_SHA384	0.73	179.4
	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	1.47	108.5
	TLS_RSA_WITH_AES_256_CBC_SHA256	0.72	187.3
	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	1.46	101.2
	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	1.47	109.7
	TLS_RSA_WITH_AES_128_GCM_SHA256	0.73	184.3
	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	1.48	108.8
	TLS_RSA_WITH_AES_128_CBC_SHA256	0.72	177.1
	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	1.47	102.3
	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	1.46	109.6
	TLS_RSA_WITH_NULL_SHA256	0.72	177.4

These measurements used the optional Crypto Express 8S Coprocessor.

Note that support for TLS 1.3 ciphers was introduced in [MQ for V9.2.0](#).

TLS 1.3 ciphers are both more expensive and slower to start than some TLS 1.2 ciphers. This is in part due to the TLS 1.3 requirement to support key shares. For the table above, ICSF FMID HCR77D1 has been applied to the performance test environment, and this results in comparable performance between TLS 1.2 and TLS 1.3 ciphers at channel start.

As discussed in the blog [“Impact of certificate key-size on TLS-protected MQ channels”](#), these channel start costs were derived using certificates with key-sizes of 2048 bits. Provided cryptographic hardware is available to assist the channel start process, the certificate key-size, whether 2048 bits or not, should have minimal impact on the cost of starting TLS-protected channels.

This is discussed further in the [SSL and TLS](#) section.

Factors affecting channel throughput and cost

- Message persistence, especially for NPMSSPEED(FAST) channels
- Message size
- Achieved batch size is very significant for both throughput and cost
 - And is often much smaller than the defined BATCHSZ channel parameter
 - You need to understand what batch size your configuration will typically achieve before using the following charts to estimate possible throughput and CPU cost.
 - See “[Determine Achieved Batch Size using MONCHL attribute](#)” which discusses how the MONCHL attribute can be used in conjunction with “DISPLAY CHSTATUS(channelName) XBATCHSZ” to determine the size of the batches achieved over the running channel.
- With a pre-loaded transmission queue you can probably achieve a batch size equal to the BATCHSZ parameter setting.
- Otherwise you can probably only achieve an average batch size < 2 with most batches consisting of just 1 message, unless you can take advantage of the BATCHINT parameter.
- Message throughput is highly dependent on the configuration used:
 - Speed and utilisation of the network
 - Response time of the IBM MQ log devices
 - CPU speeds, at both ends
 - Whether messages on the transmission queue have to be retrieved from page set
- For heavy persistent workloads we recommend CHIADAPS(30) for systems with up to 4 processors and then increase CHIADAPS by 8 for each additional processor up to a maximum of CHIADAPS(120). We have seen no significant disadvantage in having CHIADAPS(120) where this is more adapter TCBs than necessary.

SSL and TLS

The use of SSL and TLS protection on MQ channels increases the CPU cost of transmitting messages at both the sender and receiving end of the channel. The increased cost varies depending upon a number of factors and these costs are discussed in the following section.

By default, IBM MQ now supports a minimum requirement of TLS 1.2 ciphers, but for historic reasons the MQ attributes associated with encrypting messages over MQ channels are named with "SSL" in the attribute name, e.g. SSLCIPH. As such we will use terms SSL and TLS interchangeably.

When do you pay for encryption?

SSL encryption costs are incurred in the channel initiator address space at several points:

- Starting and stopping of the channel. If this occurs frequently, the cost may be a factor in choosing the cipher. Note that the key-size in the certificate may affect the rate at which channels are started - the blog "[Impact of certificate key-size on TLS-protected MQ channels](#)" discusses this impact.
- Re-negotiate secret key - the impact of this is dependent upon the value of SSLRKEYC and the volume of data flowing over the MQ channel. If small volumes of data flow over the MQ channel, or the SSLRKEYC is configured such that there are few key re-negotiations, the negotiation cost may not be a factor.
 - Note: Since TLS 1.3 has key re-negotiation as part of the protocol, reaching the SSLRKEYC threshold does not have the same effect when using a TLS 1.3 cipher compared to a TLS 1.2 cipher. There is cost associated with SSLRKEYC(non-zero) even with TLS 1.3 ciphers, and those are reported in the [key \(re-\)negotiation](#) section. For best performance with TLS 1.3 ciphers, specify SSLRKEYC(0).
- Cost of encryption and decryption of data - there are ciphers that do not have hardware support for encryption and decryption of data, which can impact the cost and the time to encrypt the message data.

The cost of TLS protection can be reduced by varying the frequency of the channel start and amount of data that flows between secret key negotiations and to a certain extent by the level of data encryption but these costs should be considered against the importance of the integrity of the data flowing over the channels.

How can I reduce the cost?

- **Change re-negotiation frequency.**

By increasing the re-negotiation frequency, the keys will be changed less often, however this means that more data will flow using the same key which in turn can give potentially more opportunity to crack the secret key.

- **Encryption level via CipherSpec**

Not all cipher specifications cost the same! Is it really necessary to use a high level of encryption with low-value data? If not, then a lower level of encryption may be appropriate.

- **Channel start/stop versus long running channels**

At channel start, the channels have to perform a handshake which includes a negotiation of secret key. If the channel is short lived and only sends a small amount of data before stopping, the secret key negotiation may be more frequent than desired.

- **Cost versus data security**

It should be determined by the company or companies using the channels whether data security is more important than the cost of encrypting the data.

- **Consider the use of channel compression**

Using COMPMSG(ZLIBFAST) when running on IBM z15 or on earlier hardware where there are zEDC features available can increase the number of messages flowing between secret key negotiations and can in certain circumstances reduce the overall MQ costs.

- **Running on the latest possible hardware**

The level of IBM processor model can make a significant difference to the cost of the MQ workload both for protected and unprotected workloads. Newer ciphers, such as TLS 1.2 have been optimised on later hardware and the cost of encryption of more secure ciphers on the latest hardware can be lower than less secure ciphers on older hardware.

- **Offloading work onto Crypto Express cards**

- The use of System SSL by IBM MQ for z/OS means that only secret key negotiations can be offloaded onto cryptographic coprocessor or accelerators.
- The data encryption and hashing is not eligible for offloading. Where possible,⁴ this is performed by CPACF's (Central Processor Assist Cryptographic Function). Data encryption and hashing will see additional CPU usage in the channel initiator address space.

⁴Some cipher specs are not supported by the CPACF. See “[Overview of hardware cryptographic features and System SSL](#)”.

Will using cryptographic co-processors reduce cost?

When cryptographic co-processors are available, they will be used for offloading the part of the TLS key processing which occurs at channel start or when the amount of data flowed over the channel reaches the SSLRKEYC setting.

TLS ciphers are able to significantly reduce key negotiation costs by offloading work onto Cryptographic co-processors and to some extent CPACF. There is still some cost impact to the MQ channel initiator address space. For example using `TLS_RSA_WITH_AES_256_GCM_SHA384` costs are typically 2-7 milliseconds per negotiation but when negotiation is entirely in software, the cost increases to 70-133 milliseconds on IBM z15 using Crypto Express 7S.

Since our systems have both CPACF and Crypto Express available, in order to determine the cost without hardware cryptographic support we force System SSL to use software by specifying the CEEOPTS DD statement in the MQ channel initiator JCL with `ENVAR("GSK_HW_CRYPTO=0")` specified.

For best performance with MQ channels using TLS cipher protection, it is advised to ensure that sufficient Cryptographic co-processors and accelerators are available.

TLS 1.3 cipher support

[MQ for V9.2.0](#) introduces support for Transport Layer Security (TLS) 1.3 protocols as well as support for aliases.

The ability to specify a TLS alias allows the queue manager and its remote partner, whether another queue manager or a client, to negotiate the ***most secure cipher*** that both partners support.

The determination of the most secure cipher is decided by System SSL on z/OS and GSKit on distributed, and by default the order is the same. The order may be different if the user sets the allowed list on at least one of the sides, in which case this can affect the cipher chosen in the determination process.

The available aliases at MQ 9.2 are as follows (in most to least secure order):

- ANY_TLS13_OR_HIGHER
- ANY_TLS13
- ANY_TLS12_OR_HIGHER
- ANY_TLS12
- ANY - *This potentially allows SSL 3 ciphers to be selected and should be used with caution.*

In order to use an alias, the MQ channel attribute “SSLCIPH” would be set to the value of the preferred alias.

Why use TLS 1.3?

The report [Enabling TLS 1.3 in System SSL Applications](#) explains in detail the benefits of the TLS 1.3 protocol and what follows is an extract of that report:

The Transport Layer Security (TLS) 1.3 protocol is a major rewrite of prior TLS protocol standards. After being in the works for many years in the Internet Engineering Task Force (IETF) TLS working group, TLS 1.3 became a formal standard in August 2018 when RFC 8446 was published. In z/OS 2.4, System SSL added support for the TLS 1.3 protocol in order for z/OS applications to take advantage of the security updates.

TLS 1.3 includes the following updates:

- All handshake messages after the initial client and server handshake messages are now encrypted. In prior protocols, messages were not encrypted until after the final handshake messages were exchanged between the client and server.
- Encrypted handshake messages are presented as payload messages and must be decrypted in order to determine whether the message is a handshake, payload or alert message.
- The RSA key exchange is no longer supported. It was replaced with Elliptic Curve Diffie-Hellman Ephemeral (ECDHE), which provides forward secrecy.
- Prior to TLS 1.3, the negotiated key exchange was part of the cipher suite. In TLS 1.3, the negotiated key exchange is no longer part of the cipher suite and is negotiated separately.

TLS 1.3 requires key sharing and MQ uses the GSK_CLIENT_TLS_KEY_SHARES option, which will generate public/private key pairs for each key share group, which can be computationally expensive and might impact performance.

MQ's current implementation of TLS 1.3 protocol support means that all of the key share groups are supported, so whether a TLS 1.3 protocol is explicitly specified, or an alias is specified that allows a TLS 1.3 cipher to be negotiated, the channel start costs *may* be significantly higher than if TLS 1.2 protocols are specified. On IBM z15, ICSF FMID HCR77D1 reduces the cost of the key share handshake to the extent that TLS 1.3 ciphers are no longer significantly more expensive than TLS 1.2 ciphers.

Using an alias in the channel definition will only affect the channel start rate and cost, i.e. the rate and cost of the channel stop, secret key negotiation and encryption/decryption of data will depend on the performance of the negotiated cipher.

The impact of the alias on channel start performance can be viewed in section "[Starting TLS channels using Aliases](#)".

Deprecated CipherSpecs

MQ version 8 saw the number of supported CipherSpecs reduced. The IBM Knowledge Center section [Enabling CipherSpecs](#) details the currently supported ciphers.

The section also details how you may re-enable deprecated ciphers on z/OS with the use of particular DD statements in the channel initiator JCL.

By default, MQ on z/OS supports TLS 1.2 ciphers.

Starting TLS channels using aliases

In the earlier section “[Starting TLS channels](#)” , it was demonstrated that the cipher used can impact the rate the channels could start as well as affect the cost of starting the channel.

By specifying an alias, such as ANY_TLS13, MQ is able to negotiate the most secure common cipher supported by both ends of the channel.

Using an alias that allows TLS 1.3 protocols to be included in the negotiation phase of channel start means that MQ must generate public/private key pairs for *each potential* key share group.

The following table demonstrates how the specified alias can impact the channel start rate and cost.
Table: Impact of alias on channel start

Alias	Negotiated Cipher	Cost CPU ms	Channels started / second
ANY_TLS13_OR_HIGHER	TLS_CHACHA20_POLY1305_SHA256	0.89	168
ANY_TLS13	TLS_CHACHA20_POLY1305_SHA256	0.88	166
ANY_TLS12_OR_HIGHER	TLS_CHACHA20_POLY1305_SHA256	0.89	164
ANY_TLS12	TLS_RSA_WITH_AES_256_GCM_SHA384	0.74	184
ANY	TLS_CHACHA20_POLY1305_SHA256	0.89	169

Note: Where a TLS 1.3 cipher *may* be negotiated, there is additional cost due to the need to generate key pairs for all key share groups. This has an impact on the rate at which the MQ channels may start.

Stopping TLS channels

In terms of stopping MQ channels, we saw TLS 1.2 and TLS 1.3 ciphers typically cost between 440 and 450 CPU microseconds per channel on IBM z16.

Note: Stopping a channel where the SSLCIPH used an alias does not affect the cost or rate at which the channel is stopped.

Secret key (re-)negotiation costs

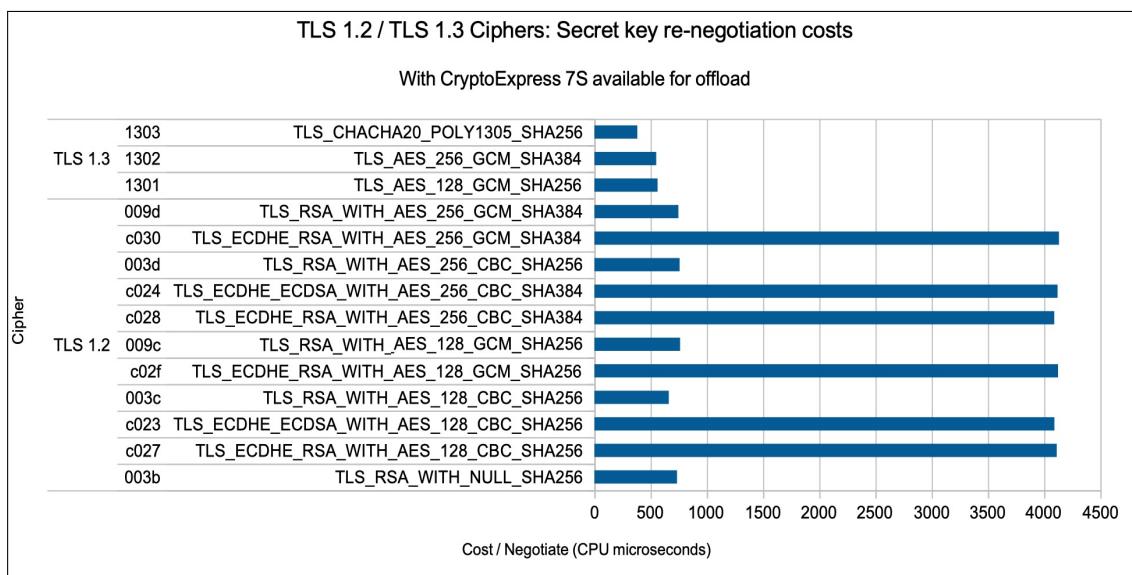
The cost of re-negotiating the secret key can vary significantly depending on the cipher used.

TLS 1.3 ciphers have key re-negotiation as part of the protocol, so the processing triggered by reaching the threshold as defined by the SSLRKEYC attribute does not perform key re-negotiation. However, even though MQ does not attempt to re-negotiate the key, there will be cost associated with a non-zero SSLRKEYC for TLS 1.3 ciphers.

Ideally, for queue managers using TLS 1.3 ciphers, the SSLRKEYC attribute would be set to 0, but as the attribute is defined at the queue manager level, this may affect any TLS channels that run using TLS 1.2 ciphers.

The following chart shows the cost of each key negotiation on IBM z15 for all of the TLS 1.3 and TLS 1.2 ciphers currently supported.

Chart: TLS Cost of secret key re-negotiation



How did we calculate the key re-negotiation costs?

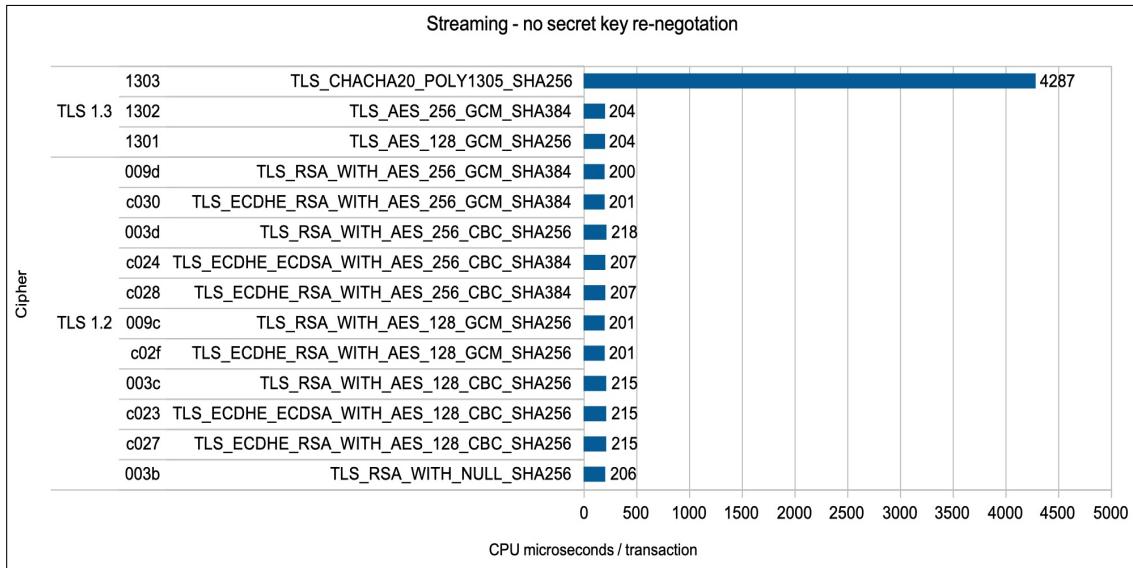
The key re-negotiation costs were calculated as follows:

- Workload run with SSLRKEYC(0) configured and the average transaction cost was determined.
- Workload run with SSLRKEYC(1MB) configured and the average transaction cost was determined.
- As we knew the message size was constant, we could determine how many messages would flow between key negotiations, taking into account that the workload is a request / reply workload.
- The difference between the 2 transaction costs multiplied by the number of messages per negotiation is the cost of re-negotiating the secret key.

Cost of Encryption / Decryption of data using TLS ciphers

The cost of encrypting and decrypting data flowing over MQ channels is typically reduced with the use of CPACF (Central Processor Assist for Cryptographic Functions) and is relatively consistent, however cipher `TLS_CHACHA20_POLY1305_SHA256` is not supported by hardware encryption and must be processed using software.

Chart: TLS Transaction cost when streaming 32KB non-persistent messages



This workload represents the transaction cost including the putting application, sending MQ subsystem, receiving MQ subsystem and getting application and includes the cost of encrypting and decrypting the messages once per transaction.

As a guide, the equivalent workload without TLS-protected MQ channels cost 115 microseconds.

Can I influence which ciphers are chosen?

Yes, from [MQ for V9.2.0](#). When using an alias to allow MQ to negotiate from a set of supported TLS ciphers, it is possible to restrict the choice of cipher to a subset of specific ciphers.

For example, whilst `TLS_CHACHA20_POLY1305_SHA256` is regarded as the most secure cipher currently available and supported on z/OS, the cost of the encryption may mean that particular cipher is less palatable from a performance perspective.

To configure the queue manager to use only a subset of the available ciphers, code the `CSQMINI` DD card in the MSTR JCL, i.e. the following sample will limit the selection of TLS 1.3 ciphers to `TLS_AES_256_GCM_SHA384` and `TLS_AES_128_GCM_SHA256`.

```
\\\CSQMINI DD *
TransportSecurity:
AllowTLSV13=TRUE
AllowedCipherSpecs=TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256
\*
```

The “`AllowedCipherSpecs`” keyword is not limited to TLS 1.3 ciphers, for example to limit the ciphers to `TLS_RSA_WITH_AES_256_CBC_SHA256` specify the contents of the `CSQMINI` DD card as:

```
\\\CSQMINI DD *
TransportSecurity:
AllowedCipherSpecs=TLS_RSA_WITH_AES_256_CBC_SHA256
\*
```

SSLTASKS

The number of SSLTASKS required depends primarily on the number of channel initiator dispatchers in use. Typically once an SSL server task processes work for a particular dispatcher and all of its channels, there will remain an affinity until restart.

There is also some benefit in ensuring that the number of SSLTASKS is greater than the number of processors in the system.

How many do I need?

This will depend on the number of dispatcher tasks specified in the channel initiator address space, but typically best performance can be achieved with CHIDISPS + 1

Why not have too many?

When a channel starts that requires SSL, the channel initiator will choose the first available SSL task for the initial handshake. For the lifetime of the channel, this same SSL task will be used.

This means that if the channels start at periods when there is no SSL work occurring, it is possible that all of the channels will be using the same small set of SSL tasks. As a result there may be idle SSL tasks in the channel initiator address space.

Each SSL server task uses 1.3MB of storage from the channel initiators available storage, which can impact the number of channels able to be started.

Why not have too few?

If too few SSL server tasks are available, then channel throughput can be constrained as the channels wait for an SSL task to become available.

SSLTASK statistics

With the introduction of Channel Initiator Accounting and Statistics in IBM MQ version 8.0.0, the usage of the SSLTASKS can be reported. This can be used as a guide to whether there are sufficient SSL tasks defined in the channel initiator.

Currently there is no simple mechanism to determine which channel is using a particular SSL task, however the report from program MQSMF⁵ as shown below, does indicate how busy the available SSL tasks are.

If a channel using SSL encryption appears to be performing worse than previously and the SSLTASK report indicates that the SSLTASKS in use are busier than 90%, restarting the channel may move which dispatcher and SSLTASK is used by that channel and result in less waiting for the SSLTASK.

MVAA,VTS1,2014/12/20,05:40:27,VRM:900,
From 2014/12/20,05:39:26.341866 to 2014/12/20,05:40:27.159296 duration 60.817430
Task,Type,Requests,Busy %, CPU used, CPU %, avg CPU , avg ET
, , , , Seconds, , uSeconds, uSeconds
0,SSL , 78284, 26.4, 3.943509, 6.6, 50, 202
1,SSL , 37, 0.0, 0.000213, 0.0, 6, 6
2,...
8,SSL , 37, 0.0, 0.000210, 0.0, 6, 5
9,SSL , 367304, 27.0, 4.429724, 7.4, 12, 44

Note: In the example MQSMF report, there are 10 SSLTASKS available, of which task 0 and 9 are in use and both of these tasks have capacity to support more channels.

⁵Program MQSMF is available as part of supportPac MP1B “Interpreting accounting and statistics data”

SSL channel footprint

Typically SSL uses an additional 30KB for each channel but for messages larger than 32KB can be higher.

SSL over cluster channels

Using SSL over cluster channels should be no more expensive than SSL over non-cluster channels.

Note: Repository information being shared will flow over channels and will contribute to the amount of data flowing over a channel and may cause the secret key renegotiation to occur earlier than expected.

Similarly this repository information will be encrypted across the network and will be subject to additional encryption and decryption costs.

SSL over shared channels

Shared channels will update a DB2 table when their state changes. This can result in shared channels taking longer to start than non-shared channels. No SSL-state information like SSLRKEYS is held in DB2 so using SSL over shared channels will not affect the performance of the channels once the channel is started.

Note: When using shared channels, the channel initiator will check for a certificate named `ibmWebSphereMQ<QSG>` and then `ibmWebSphereMQ<QueueManager>`. This allows the user to use a single certificate for their entire queue sharing group. IBM MQ version 8.0 allows these certificates to be overridden by the certificate named in the CERTABL channel parameter.

Using AT-TLS to encrypt data flowing over IBM MQ channels

Application Transparent Transport Layer Security (AT-TLS) is based on z/OS System SSL, and transparently implements the TLS protocol (defined in RFC 2246) in the TCP layer of the stack.

When running channels between queue managers hosted on z/OS, AT-TLS can be used to encrypt messages transported over MQ channels rather than relying on IBM MQ channels performing the encryption function. The use of AT-TLS can result in reduced costs within MQ.

It should be noted that whilst AT-TLS can offer reduced costs, these are not always reflected in improvements in the rate that message data is transferred, particularly when using encryption ciphers that are unable to exploit hardware acceleration, such as TLS 1.3 cipher `TLS_CHACHA20_POLY1305_SHA256`.

Who pays for AT-TLS

MQ channels with `SSLCIPH` configured will see the encryption/decryption cost associated with the channel initiator address space.

When transporting messages using channels encrypted using AT-TLS, the cost of encryption is charged to the callers unit of work (i.e. the channel initiator) and decryption is incurred by the TCPIP address space as the decryption is performed by an SRB running in the TCPIP address space.

Limitations

IBM MQ allows the user to specify different SSL cipher specifications for each channel.

To run with different cipher specifications using AT-TLS can involve defining additional rules plus either specifying the `LOCLADDR` attribute on the channel to restrict the port being used or by running with multiple listeners defined on the channel initiator.

IBM MQ allows the secret key negotiation to be performed when the number of bytes sent and received within an SSL conversation exceeds the `SSLRKEYC` value, whereas AT-TLS allows the renegotiation to take place after a period of time has been exceeded.

When the AT-TLS encryption is performed, the TCP socket is blocked – this can have a noticeable effect on throughput rate with large messages unless dynamic right sizing is enabled on the TCPIP stack.

Channels protected with `CHLAUTH` rules may not be allowed to start if the rule contains a value for `SSLPEER`.

Performance comparison

The following measurements were run using a 10Gb network between 2 z/OS v2r5 LPARs each with 3 dedicated processors on an IBM z16 (3931-7K0).

A request/reply workload between 2 queue managers was run over a pair of sender-receiver channels using non-persistent messages.

In the measurements using channels with `SSLCIPH` cipher specifications, the SSL key negotiation has been disabled (by setting to '0') in order to provide a direct comparison. Similarly the AT-TLS negotiation period has been disabled.

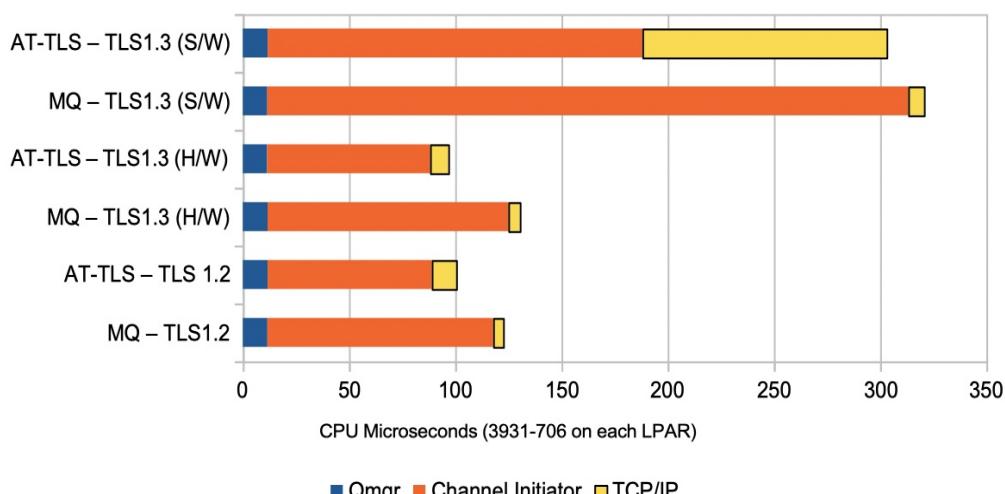
The costs shown in the following charts are for the queue manager, channel initiator and TCPIP address spaces only and are based on the cost observed in both LPARs.

For the purposes of these measurements, the following ciphers have been used:

Type	MQ SSL/TLS cipher	AT-TLS cipher
TLS 1.2 (hardware - CPACF)	TLS_RSA_WITH_AES_256_CBC_SHA256	TLS_RSA_WITH_AES_256_CBC_SHA
TLS 1.3 (hardware - CPACF)	TLS_AES_256_GCM_SHA384	TLS_AES_256_GCM_SHA384
TLS 1.3 (software)	TLS_CHACHA20_POLY1305_SHA256	TLS_CHACHA20_POLY1305_SHA256

As discussed in the [MQ for V9.2.0](#) performance report, the cost of encrypting and decrypting data flowing over MQ channels is typically reduced with the use of CPACF (Central Processor Assist for Cryptographic Functions) and is relatively consistent. However cipher **TLS_CHACHA20_POLY1305_SHA256** is not supported by hardware encryption and must be processed using software, hence for the purposes of these measurements, we are calling the cipher “TLS 1.3 (software)”.

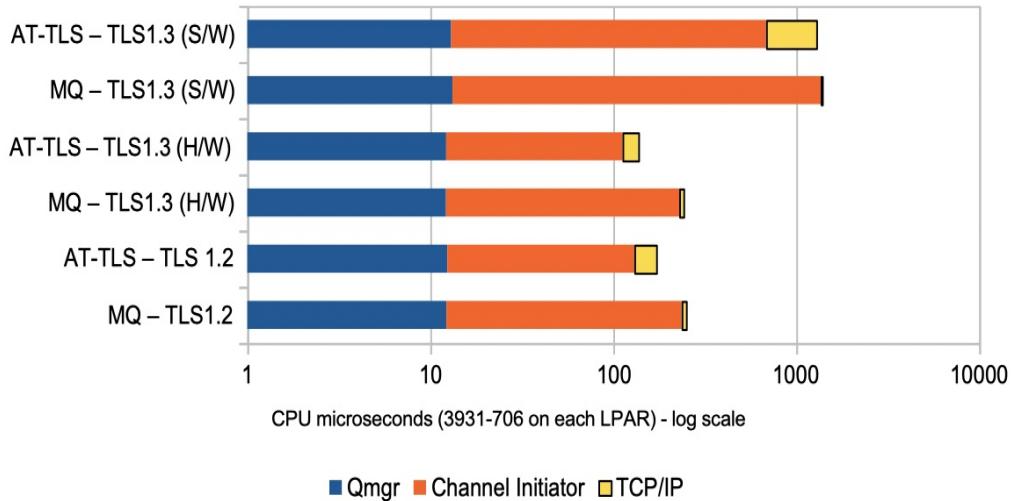
Request/Reply transport cost using 2KB non-persistent messages



Notes on preceding chart:

- Cost of transporting the message is up to 25% lower when using AT-TLS to protect the message, depending on cipher used.
- Channel initiator costs are up to 41% lower when using AT-TLS.
- TCPIP costs are 1.6 times higher when using ciphers able to exploit hardware encryption with AT-TLS.
- TCPIP costs are 15 times higher when using ciphers that rely on software encryption with AT-TLS when compared with equivalent TCPIP costs on MQ channels protected using SSLCIPH.

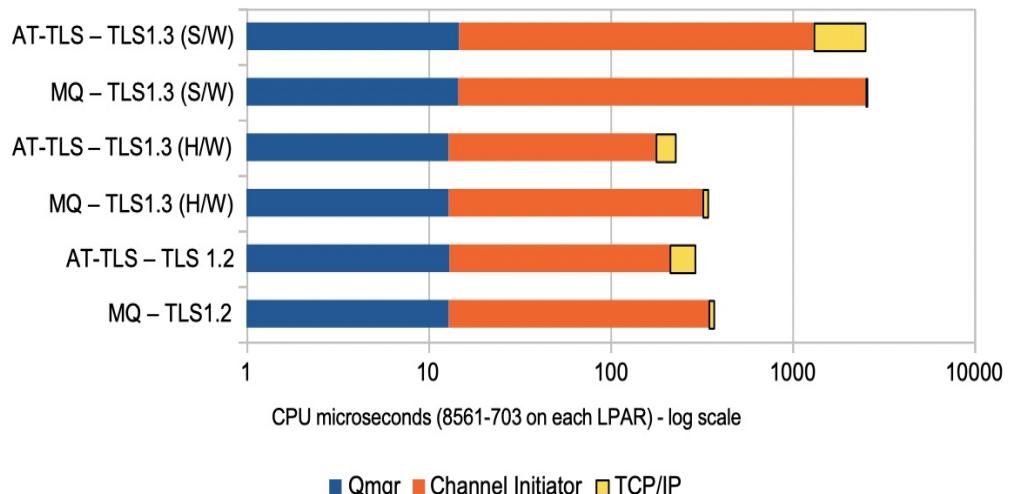
Request/Reply transport cost using 16KB non-persistent messages



Notes on preceding chart:

- Chart uses a log scale - for the AT-TLS using TLS 1.3 (software) the costs attributed to the channel initiator and the TCP/IP address space are approximately equal at 1.2 CPU milliseconds per transaction.
- Cost of transporting the message is up to 43% lower when using AT-TLS to protect the message, depending on cipher used.
- Channel initiator costs are up to 54% lower when using AT-TLS.

Request/Reply transport cost using 32KB non-persistent messages

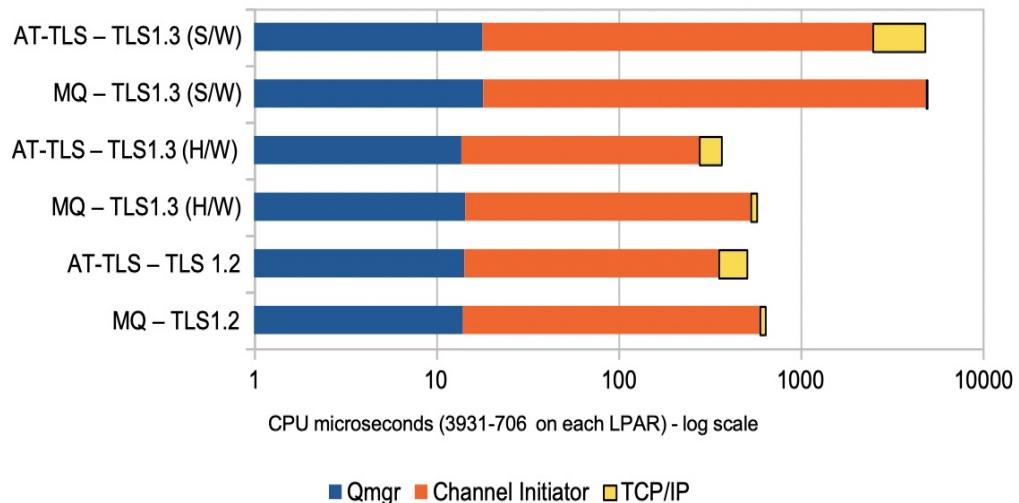


Notes on preceding chart:

- Chart uses a log scale - for the AT-TLS using TLS 1.3 (software) configuration, the costs attributed to the channel initiator and the TCP/IP address space are approximately equal, at 2.3 CPU milliseconds per transaction.

- Cost of transporting the message is up to 34% lower when using AT-TLS to protect the message, depending on cipher used.
- Channel initiator costs are up to 48% lower when using AT-TLS.

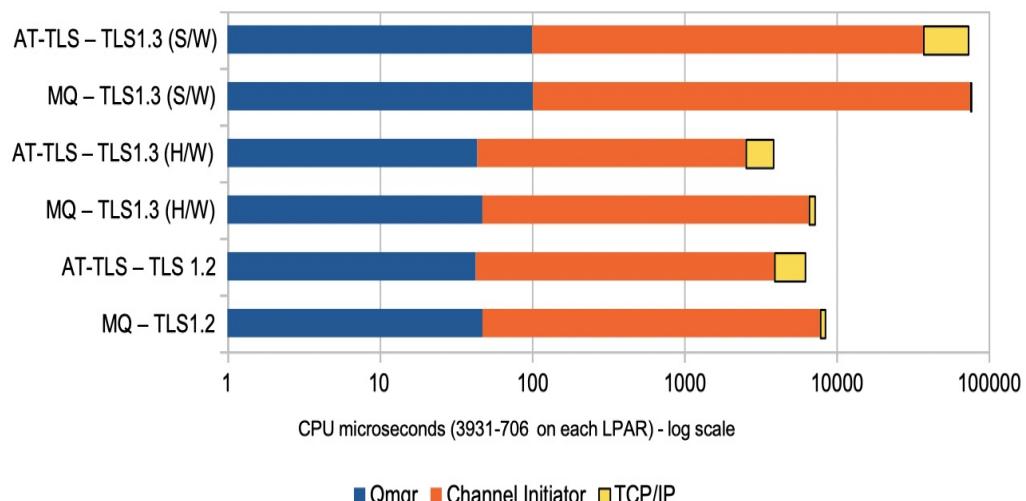
Request/Reply transport cost using 64KB non-persistent messages



Notes on preceding chart:

- Chart uses a log scale - for the AT-TLS using TLS 1.3 (software) configuration, the costs attributed to the channel initiator and the TCP/IP address space are approximately equal, at 4.8 CPU milliseconds per transaction.
- Cost of transporting the message is up to 36% lower when using AT-TLS to protect the message, depending on cipher used.
- Channel initiator costs are up to 50% lower when using AT-TLS.

Request/Reply transport cost using 1MB non-persistent messages



Notes on preceding chart:

- Chart uses a log scale - for the AT-TLS using TLS 1.3 (software) configuration, the costs attributed to the channel initiator and the TCP/IP address space are approximately equal, at 73 CPU milliseconds per transaction.
- Cost of transporting the message is up to 46% lower when using AT-TLS to protect the message, depending on cipher used.
- Channel initiator costs are up to 62% lower when using AT-TLS.

Is the reduced cost reflected in a throughput improvement?

As the previous measurements have shown, there is potential for reduced CPU cost when transporting MQ data over networks protected by AT-TLS encryption rather than protecting MQ channels using the SSLCIPH channel attribute.

Those cost differences vary depending on the message size and the cipher used. In the measurements reported previously, the cost differences were:

- TLS 1.2 (hardware) - between 18-31% lower cost using AT-TLS.
- TLS 1.3 (hardware) - between 25-47% lower cost using AT-TLS.
- TLS 1.3 (software) - between 2.7-7% lower cost using AT-TLS.

When hardware can be exploited, we observed a cost saving from AT-TLS rather than MQ and System SSL services, but this saving is negligible when using ciphers that have to be processed in software cycles.

In terms of transaction rates, the benefits are less clear - generally the round-trip times are similar when using ciphers able to exploit hardware encryption, and using these ciphers the workloads are constrained by network limits. It is worth noting that the encryption/decryption of data when using AT-TLS is on the same TCPIP thread as the network send/receive request. This can lead to network sizing windows to be less optimal than when using MQ with System SSL.

Additionally, with regards to TLS 1.3 cipher TLS_CHACHA23_POLY1305_SHA256, the AT-TLS throughput degraded relative to the MQ with System SSL configuration, particularly as the message size increased.

Table: Comparing the average transaction round-trip times (milliseconds) using TLS 1.3 (software) cipher TLS_CHACHA23_POLY1305_SHA256:

Message Size	MQ + System SSL	MQ + AT-TLS	% increase in round-trip time
2KB	1.04	1.00	-4
16KB	3.96	3.83	-3.3
32KB	7.10	7.01	-2.2
64KB	13.36	13.90	+4
1MB	202.47	202.57	+0.05

In the 1MB measurement using the AT-TLS with cipher TLS_CHACHA23_POLY1305_SHA256, the measurement was able to achieve the same XBATCHSZ as the MQ with System SSL configuration, but saw a significant increase in NETTIME, rising from between 1-4 milliseconds up to 5-10 milliseconds, due to MQ's collection of NETTIME data including AT-TLS' encryption and decryption of the data.

Why is there no improvement to transfer rate despite the transport cost being reduced?

The model used by Comms Server and AT-TLS to protect the data is a simple one - an SRB per socket processes the request - this processing involves both the encryption and sending of the data. As a result the send of the data over the network cannot begin until the encryption of the data is complete.

Similarly at the receiving-side, the SRB will receive the data and decrypt it, before becoming available to receive more data.

With the extended time that the encryption and decryption takes when using AT-TLS, particularly with cipher `TLS_CHACHA23_POLY1305_SHA256` with larger payloads, the SRB is blocked for longer periods.

By contrast, MQ has dedicated tasks for the network interaction (dispatchers) and separate tasks for the encryption workload (SSL tasks). This enables MQ to be able to run encryption of one message on an SSL task in parallel with sending another message (or indeed chunk of a large message) over the network using a dispatcher task.

Starting and stopping MQ channels protected by AT-TLS

The rate and cost at which MQ channels are able to start and stop is particularly of interest where the channels are starting and stopping regularly, perhaps due to a sudden increase or decrease in workload.

Both this document in section “[Channel start and stop rates and costs](#)” and the [MQ for V9.2.0](#) performance report provide example costs and achieved rates when starting and stopping MQ channels, with and without MQ ciphers.

The cost of starting MQ channels with TLS 1.3 ciphers has been reduced since the initial release of MQ for z/OS v9.2 due to several reasons:

- A substantial performance improvement in ICSF FMID HCR77D1 which reduces the cost of the key share handshake.
- MQ reduces the number of client key shares used - this code is only available in MQ for z/OS CD releases v9.2.2 onwards.

The measurements relating to channel start and stop performance in this section use the MQ for z/OS 9.3 code base.

The information in this section is based upon a comparison of the starting and stopping of 4000 outbound (sender) MQ channels, that have been defined between two z/OS queue managers hosted on separate LPARs of a single IBM z16. The distance between the channel end-points will affect the rate at which a channel can start, as a high latency network will increase the time taken for the handshake process, involving multiple flows between channel initiators, to complete.

The channels are configured as defined below:

Type	MQ SSL/TLS cipher	AT-TLS cipher
TLS 1.2 (hardware - CPACF)	TLS_RSA_WITH_AES_256_CBC_SHA256	TLS_RSA_WITH_AES_256_CBC_SHA
TLS 1.3 (hardware - CPACF)	TLS_AES_256_GCM_SHA384	TLS_AES_256_GCM_SHA384
TLS 1.3 (software)	TLS_CHACHA20_POLY1305_SHA256	TLS_CHACHA20_POLY1305_SHA256

The AT-TLS configuration used in these measurements specifies the server-side `HandshakeRole serverWithClientAuth` in the `TTLSEnvironmentAction` statement.

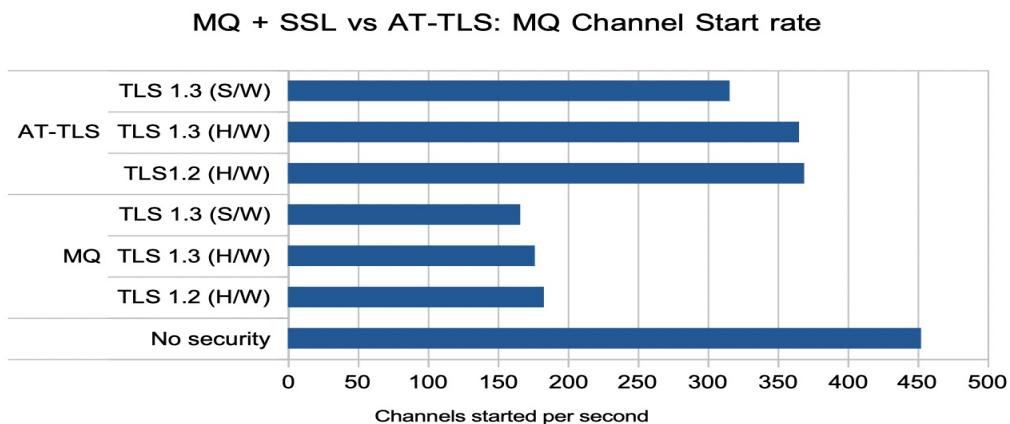
Additionally, the channels protected using AT-TLS ciphers are configured to use default options in the `TTLSEnvironmentAdvancedParms` statement for:

- `Renegotiation`
- `RenegotiationIndicator`
- `RenegotiationCertCheck`

Enabling certificate checking and full or abbreviated renegotiation may affect the costs observed.

AT-TLS start channel performance

Chart: Start channel rate

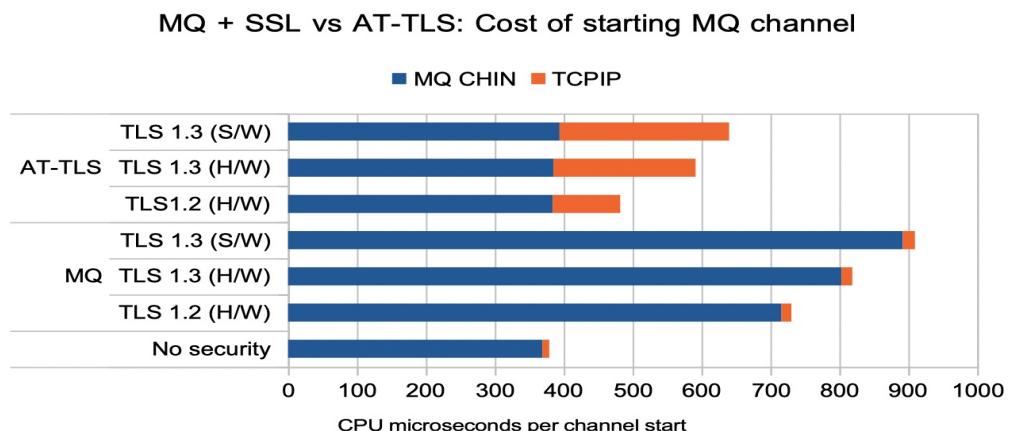


Notes on preceding chart:

Enabling secure channels has an immediate impact on the rate at which MQ is able to start the channels:

- Channels secured using MQ with System SSL are able to start at approximately 40% of the rate of unsecured channels.
- Channels secured using AT-TLS are able to start at between 70 to 80% of the rate of unsecured channels.

Chart: Start channel cost



Notes on preceding chart:

With secure channels, MQ using System SSL does see a small increase in TCPIP costs but the additional cost is primarily in the channel initiator address space.

- For the TLS 1.2 cipher used, this results in the channel initiator cost doubling that of an unsecured channel.
- For the TLS 1.3 cipher where hardware support is available, the channel initiator cost is 2.2 times that of the unsecured channel.

- For the TLS 1.3 cipher without hardware support, the channel initiator cost of the channel start is 2.4 times that of the unsecured channel.

With channels secured using AT-TLS, the impact is primarily in the TCPIP address space, and in terms of CPU cycles is typically far less than those observed in the MQ channel initiator.

- For the TLS 1.2 cipher, the TCPIP cost increases from 10 to 100 microseconds.
- For the TLS 1.3 cipher with hardware support, the TCPIP cost increases from 10 to 205 microseconds.
- For the TLS 1.3 cipher without hardware support, the TCPIP cost increases from 10 to 245 microseconds.

It is also important to consider that with the current configuration, MQ is able to offload some of the cryptographic processing to the CryptoExpress hardware (both co-processor and accelerator) at channel start, whereas the AT-TLS configuration is not using cryptographic hardware.

AT-TLS stop channel performance

Costs attributed to the MQ channel initiator for stopping channels are of the order:

- 340 CPU microseconds for unprotected channels, or channels protected by AT-TLS policies.
- 450 CPU microseconds for channels protected by System SSL.

TCPIP costs when stopping MQ channels were of the order:

- 10-15 CPU microseconds per channel with not protected by AT-TLS policies.
- 30-60 CPU microseconds when protected by AT-TLS policies.

Should I use AT-TLS to provide encryption of my MQ channels?

Using AT-TLS to provide encryption of MQ channels may offer a substantial reduction in CPU cost in the MQ channel initiator when compared with using MQ's own implementation using System SSL, and this typically results in a net reduction when including the additional CPU cost in the TCPIP address space.

With regards to the processing model that TCPIP uses, i.e. a single task per socket which encrypts and sends data, or receives and decrypts the data, despite the cost reduction there is not always a commensurate improvement in throughput. There are instances when MQ is able to exploit parallel processing when sufficient SSL tasks and dispatchers are available such that MQ can achieve parity or indeed higher throughput over the channels.

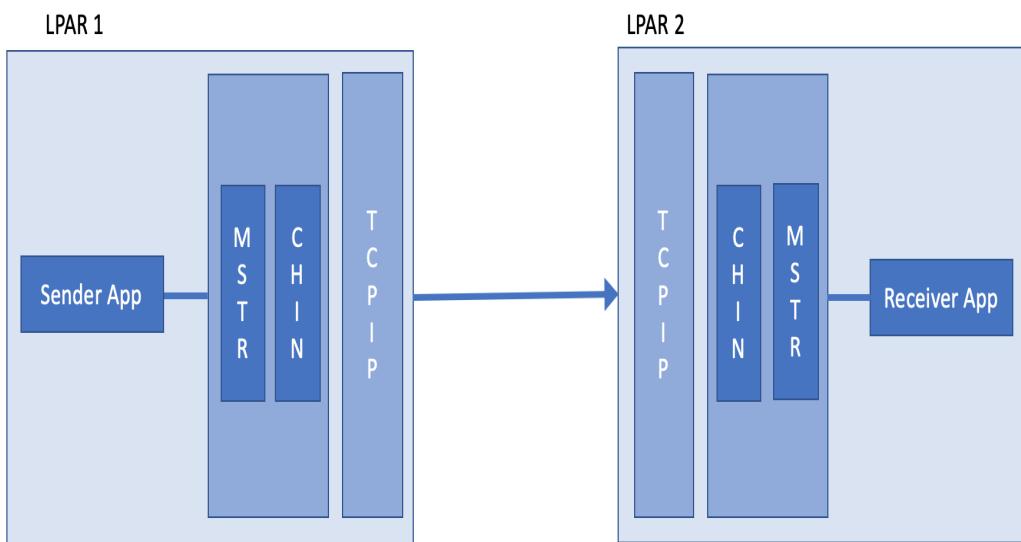
Using AT-TLS does allow the user greater control over the security settings implemented but as a result, does require further understanding of the ramifications whether choosing particular options or indeed of relying on the defaults.

However, since AT-TLS is not fully integrated with MQ, this can prevent customers from using various MQ security controls and can make configuring some security scenarios more difficult.

Costs of Moving Messages To and From z/OS Images

This section considers the total CPU costs of moving messages between queue managers in separate z/OS images. A driver application attached to a queue manager in LPAR 1 puts a message to a remote queue which is defined on a queue manager running in LPAR 2. A server application in LPAR 2 retrieves any messages which arrive on the local queue.

No code page conversion costs are included in any “MVS to MVS” measurements. See “[How much extra does code page conversion cost on an MQGET?](#)” for an estimate of typical MQFMT_STRING code page conversion costs.



Notes on diagram:

- Each z/OS image was a 3-CPU logical partition (LPAR) of an IBM z16 (3931-7K0).
- The MVS systems were connected via a 10Gb Ethernet network.
- The driver application continually loops, putting messages to the remote queue but ensures queue has maximum queue depth of 1.
- The server application continually loops, using get-with-wait to retrieve the messages.
- Neither application involves any business logic.
- The server application runs non-swappable.
- The queue is not indexed.
- All non-persistent messages were put out of syncpoint by the sender application and got out of syncpoint by the server application.

Measurements were made with two different channel settings:

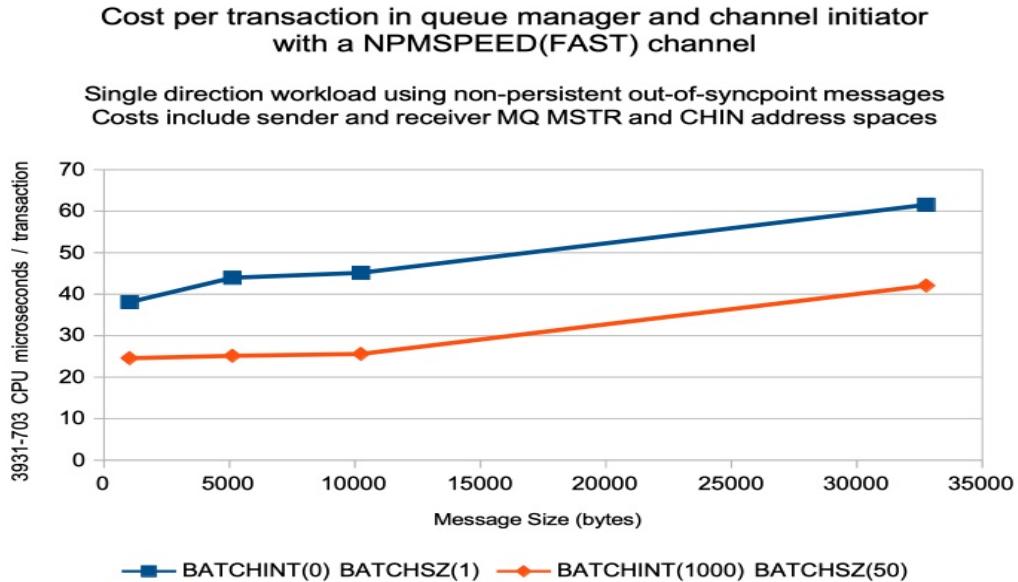
- BATCHSZ(1) with BATCHINT(0) and NPMSPED(FAST)
- BATCHSZ(50) with BATCHINT(1000) and NPMSPED(FAST)

As mentioned in the “[Tuning channels - BATCHSZ, BATCHINT and NPMSPED](#)” section, if all messages flowing over the channel are guaranteed to be non-persistent and the channel is running over a high-latency network, it may be of benefit to set the BATCHINT and BATCHSZ to higher

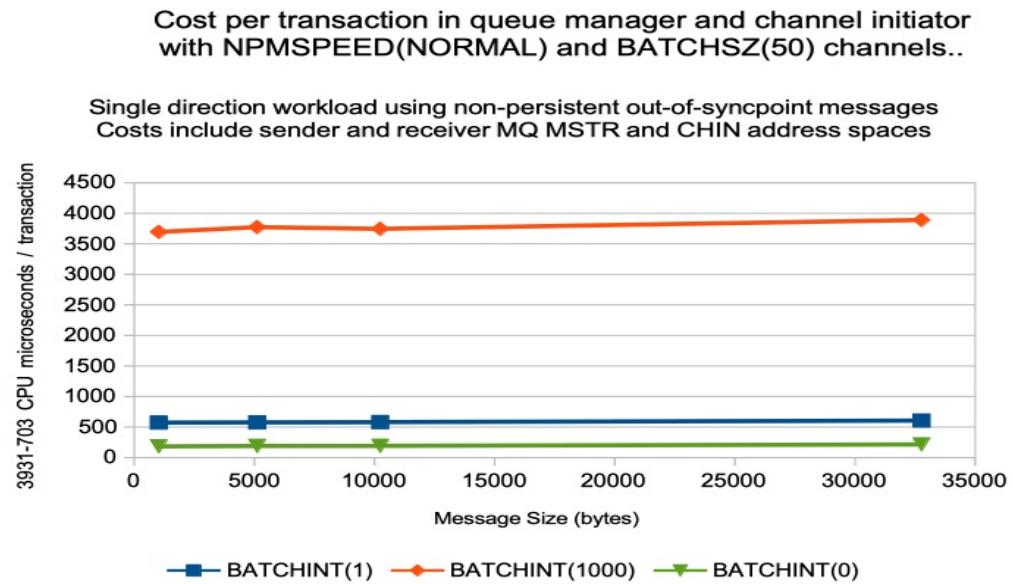
values. In our low-latency system, we set BATCHINT(100,000) and BATCHSZ(5000) and saw similar performance (cost and throughput) as the BATCHINT(1000) BATCHSZ(50) configuration.

The chart below show the CPU costs in both systems for non-persistent messages with a variety of message sizes for the queue manager and channel initiator address spaces.

NOTE: In the following 2 charts, there is only 1 requester and 1 server application, but by setting the BATCHINT to 1000 to keep the batch open for longer, the cost is significantly reduced.



Contrast the previous chart with the following that shows using a channel with NPMSPEED(NORMAL). With NPMSPEED(NORMAL) the effects of holding the batch open with BATCHINT and the cost of channel syncpoints becomes clear.



NOTE: With NPMSPEED(NORMAL), the achieved batch size was 1, but varying the BATCHINT to 0 made a significant difference to the transaction rate. For example:

- With 1 requester and a BATCHINT(1000), only 1 message per second is flowing over the sender channel.
- With NPMSPEED(NORMAL) and BATCHINT(0), the rate increased to 1460 messages per second.

Non-persistent messages - NPMSPEED(FAST)

For **BATCHSZ(1)** on IBM z16, the CPU usage is approximately:

The total (sender and receiver queue manager and channel initiator) cost:

$(38.53 + 0.0007S)$ CPU microseconds per transaction,
where S is the size of the message expressed in bytes.

The costs are shared evenly between the sender and receiver end.
E.g. for a 10 KB message this is:

$38.53 + (0.0007 * 10240) = 45.7$ CPU microseconds - approximately 22.85 at the sender end and 22.85 at the receiver end.

For **BATCHSZ(50)** on IBM z16, the CPU usage is approximately:

The total (sender and receiver queue manager and channel initiator) cost:

$(22.16 + 0.00058S)$ CPU microseconds per message
where S is the size of the message expressed in bytes

E.g. for a 10KB message this is:

$22.16 + (0.00058 * 10240) = 28.1$ CPU microseconds - approximately 14 CPU microseconds at the sender end and 14 at the receiver end.

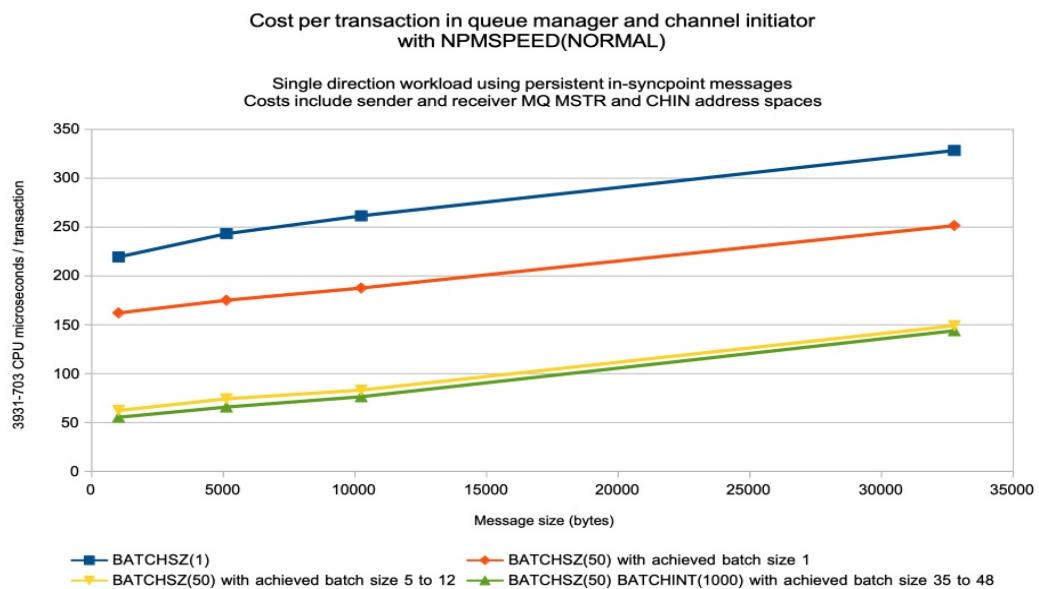
These algorithms produce figures which are within 10% of the measured figure for the range shown in the charts.

Persistent messages

All persistent messages were put within syncpoint by the sender application and got within syncpoint by the server application. Measurements were made with four different configurations all using NPMSPEED(NORMAL):

- BATCHINT(0), BATCHSZ(1)
- BATCHINT(0), BATCHSZ(50) with achieved batch size of 1.
- BATCHINT(0), BATCHSZ(50) with achieved batch size between 5 and 12.
- BATCHINT(1000), BATCHSZ(50) with achieved batch size of 35 to 48.

The chart below show the total CPU usage in both systems for persistent messages with a variety of message sizes.



A reduction in the CPU usage at both ends of the transaction when conditions allow a batch size greater than 1 can be achieved. A slow rate of MQPUT with BATCHSZ(50) will see the achieved batch rate drop to 1 and the associated cost per transaction increase to parity with BATCHSZ(1). Reducing the number of batches results in a reduction in the number of channel synchronisations and TCPIP transactions per message.

Note: That this reduction in CPU is dependent upon messages being put at a suitable rate. In a more realistic situation where the achieved batch size is close to 1, the CPU usage increases, as shown by the top line in the chart.

The lowest CPU costs are achieved when the application unit of work exactly matches the channel BATCHSZ or the BATCHINT parameter.

For capacity planning purposes it is safest to assume BATCHSZ(1) will be used. If it is known that a higher batch size can consistently be achieved, a higher figure may be used.

An approximate straight line algorithm is presented here for the CPU usage with **BATCHSZ(1)** on IBM z16:

The total (sender and receiver queue manager and channel initiator) cost:

$(223 + 0.003S)$ CPU microseconds per message
where S is the size of the message expressed in bytes

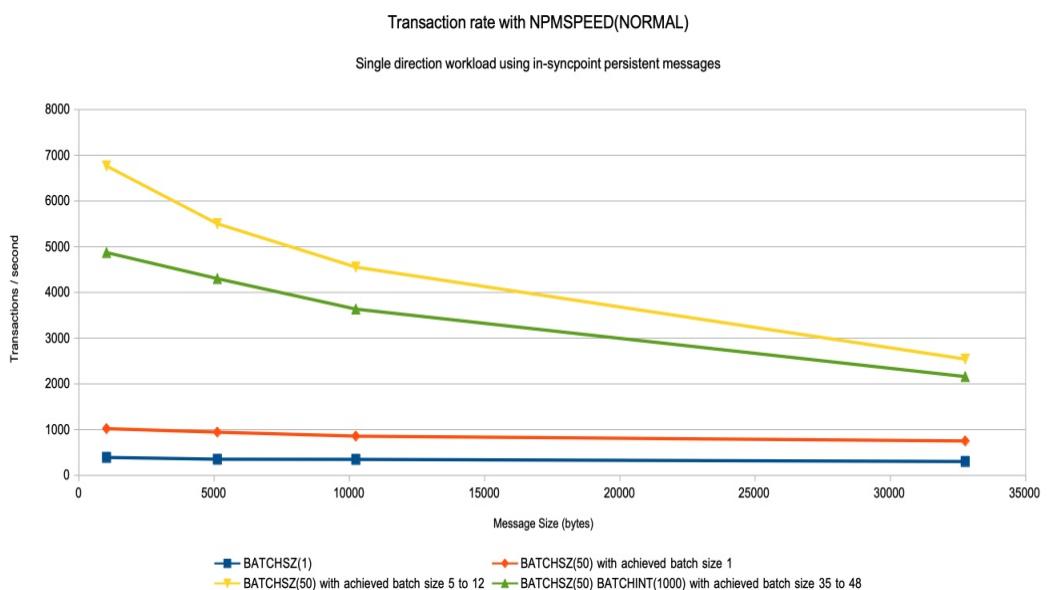
e.g. for a 10KB message this is:

$223 + (0.003 \times 10240) = 253.7$ CPU microseconds – with approximately 128 at the sender end and 124 at the receiver end.

This represents the ‘best choice scenario’ for persistent messages where the achieved batch size is undetermined.

This algorithm produces figures which are within 10% of the measured figure for the range shown in the charts.

The chart below shows the achieved transaction rate using persistent messages with a variety of message sizes.



Note: The best performance was obtained when the batch was held open for the minimum amount of time for the available work. The BATCHSZ(50) measurement that achieved batch size of 5 to 12, was able to complete the batch with the 12th message, whereas the measurement with BATCHINT(1000) held the batch open for 1 second in an attempt to try to achieve the target batch size of 50. As there was insufficient work to achieve the full batch, the end of batch flow occurred at BATCHINT.

Chapter 4

System

Hardware

DASD

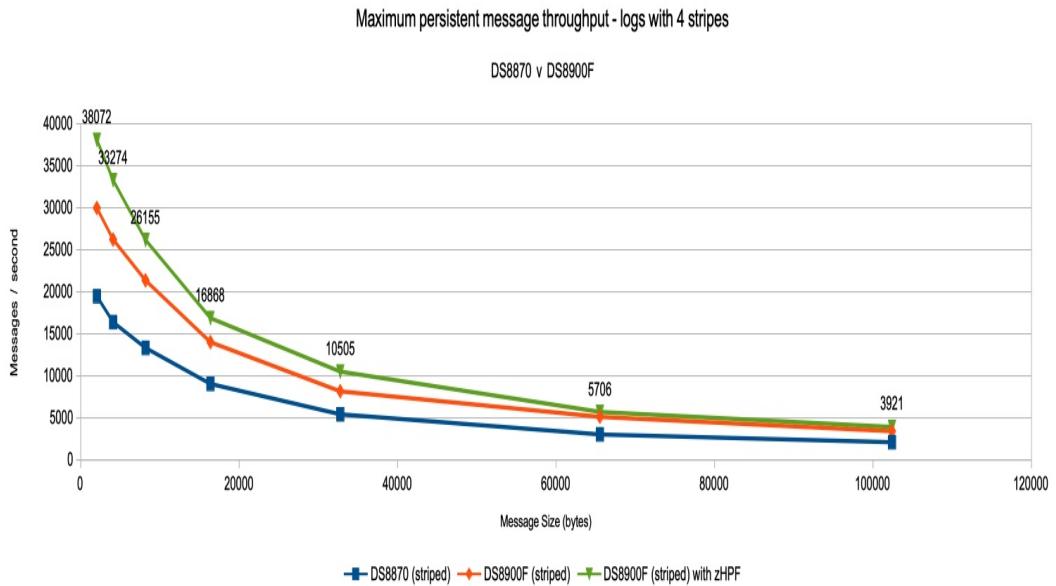
There are limits to persistent message rates achievable

- Because of upper bounds to the overall data rate to a IBM MQ log, see “[Upper bound on persistent message capacity - DASD log data rate](#)” to overcome any such limit then either faster DASD or more queue managers will be required.
- Because of upper bounds to the log I/O rate achievable by a single application instance, which may be required where messages must be processed in strict order.
- There is a limit to the maximum number of messages through one queue, see [Maximum throughput using non-persistent messages](#).

Maximum request/reply throughput (DS8900F)

The following chart shows the maximum throughput we could achieve with many request and reply applications for a range of message sizes. Sufficient processors were available so they were not a limiting factor in the throughput capacity.

As can be seen in the following chart, the underlying disk technology can make a significant difference in the rate that MQ is able to log messages.



Upper bound on persistent message capacity - DASD log data rate

The maximum total IBM MQ system capacity for persistent messages is bounded by the maximum data rate sustainable to the DASD subsystem where the IBM MQ logs reside. For IBM MQ with dual logging, the maximum sustainable data rate **for messages put at one message per commit and got at one message per commit** is about:

Log data set DASD type	1KB messages	5KB messages	1 MB messages
RVA-T82	2.3 MB/sec	2.8MB/sec	
ESS E20 with ESCON	5.3 MB/sec	7.1 MB/sec	
ESS F20 with ESCON	7.1 MB/sec	10.2 MB/sec	11.3 MB/sec
ESS F20 with FICON	7.4 MB/sec	13.0 MB/sec	15.6 MB/sec
ESS 800	10.0 MB/sec	16.5 MB/sec	24 MB/sec
DS8000 (RAID 5)	14.9 MB/sec	36.0 MB/sec	68.1 MB/sec
DS8000 (RAID 10)	15.6 MB/sec	36.1 MB/sec	68.6 MB/sec
DS8800 (RAID5)	32.1 MB/sec	62.4 MB/sec	128.7MB/sec
DS8870 (RAID 5) with 4-stripes	47.8 MB/sec	98.5 MB/sec	215 MB/sec
DS8870 (RAID 5) with 4 stripes plus zHPF enabled	64.7 MB/sec	136.4 MB/sec	337.5 MB/sec
DS8900F (RAID 5) with 4 stripes	95 MB/sec	193 MB/sec	404 MB/sec
DS8900F (RAID 5) with 4 stripes plus zHPF enabled	112 MB/sec	243 MB/sec	445 MB/sec

Note: Performance data from DASD types prior to DS8870 are included for comparison purposes, however the measurements are not directly comparable as they span different levels of IBM MQ and System Z hardware.

These are the peak rates on our system configuration. It is usually prudent to plan not to exceed 50% of the rates attainable on the target system particularly where message response time and not just overall message throughput is important.

What is the effect of dual versus single logging on throughput?

On our system use of single logging rather than dual logging increased throughput from between 5% for small messages to 50% for 1MB messages. On a system with significant DASD constraint, use of single logging might enable a much more significant throughput increase, for example on DS8870 where we saw cache contention, the single logging was able to log at up to 80% higher. Use of single logging would mean there is a single point of failure possibility in your system.

Will striped logs improve performance?

Switching to active logs which use VSAM striping can lead to improved throughput in situations where performance is being constrained by the log data rate. The benefit obtained from using VSAM striping varies according to the amount of data being written to the log on each write. For example, if the log dataset has been set up with 4 stripes, a log write carrying a small amount of data such that only one stripe is accessed will gain no benefit at all, while a log write carrying sufficient data to access all 4 stripes will gain the maximum benefit.

The increased logging rate achieved by using striped active logs will result in the log filling more quickly. Prior to IBM MQ version 8.0.0 however, the time taken to archive a log dataset is unchanged. This is because archive log datasets must not be striped as the BDAM backwards reads required during recovery are not supported on striped datasets. Thus the possibility of needing to reuse a log dataset before its previous archive has completed is increased. It may therefore be necessary to increase the number or size of active log datasets when striping is used. **If you attempt to sustain these maximum rates to striped logs for long enough then eventually you will fill all your active logs with consequent unacceptable performance.**

Version 8.0.0 saw the changing of archive datasets from BDAM to QSAM, which allows the allocation to exceed 65,535 tracks and the striping of the archive logs. Striping of the archive logs may result in an improved rate of offload. We did see an increase in cost in the queue manager address space when using archive datasets larger than 65,535 tracks when the dataset is using extended format.

In summary, striped logs are most likely to be of use where there is a reasonably predictable amount of large messages in a particular time period such that the total amount of data to be logged does not cause the active logs to be filled.

Should MQ for z/OS use log striping?

There is an article from Db2 for z/OS named “[Db2 log striping recommendation](#)” that suggests on modern DASD, Db2 customers are discouraged from striping Db2 logs with today’s disk subsystems.

In our measurements on DS8900F, striped MQ logs were able to sustain log rates of up to 50% higher than non-striped logs when using messages of 64KB or larger.

Will striped logs affect the time taken to restart after a failure?

The recovery process will need to read active logs and this is significantly quicker with striped datasets, particularly for the backward recovery phase. It may also involve reading archived log datasets that cannot be striped. Thus any use of archive log datasets during recovery will not be quicker. It is possible to minimise or even eliminate the possibility of an archive log being required during recovery. This requires pageset and if using shared queue, CF structure backup, at appropriate intervals and appropriate reaction to any CSQJ160I messages concerning long running units of recovery with a STARTRBA no longer in active logs. With version 6, implementation of log shunting, archive log datasets will not be used for recovery unless pageset or CF structure media recovery is required.

Benefits of using zHPF with IBM MQ

What is zHPF?

High Performance FICON for System z (zHPF) is a data transfer protocol that is optionally employed for accessing data from IBM DS8000 storage and other subsystems and was made available for System z in October 2008.

zHPF may help reduce the infrastructure costs for System z I/O by efficiently utilizing I/O resources so that fewer CHPIDs, fibers, switch ports and control units are required.

When can it help with IBM MQ work?

zHPF is concerned with I/O to DASD, so queue managers that are used for non-persistent workload may see little benefit.

Similarly, where the LPAR is running with low volume I/O, it is unlikely that using zHPF will give much benefit.

Where multiple queue managers exist on the same LPAR and are processing high volume persistent workload, whether local or shared queue, zHPF may be able to provide an increase in the throughput rate.

- Where throughput was restricted by the number of I/O channels available and then zHPF was enabled, we saw a doubling of throughput.
- Where throughput was restricted by the DASD logging rate, enabling zHPF saw a 17% increase in transaction rate.
- With larger messages (1MB and larger) using zHPF and striped logs, throughput was increased by 55%.

Network

The measurements run in this document have been typically run on our dedicated performance network which is rated at 1Gb.

When we moved onto a dedicated 10Gb performance network, we saw a number of changes:

- Measurements were more consistent.
- Response times for measurements with low numbers of channels were slightly faster than when run on the slower network.
- CPU became the constraint point rather than network.
- Dedicated CHPIDs on LPARs that perform high volume network traffic gave significantly better performance than shared CHPIDs – even when RMF shows CHPID is not at capacity.

By moving onto a more modern network with greater capacity we haven't greatly improved a single channels peak transmission rate, but we have been able to increase the number of channels driving data to exploit the additional band-width.

Our network is high-bandwidth low-latency which means we can expect good response times for request/reply type workloads.

It is important to consider what sort of network you have and tune your channels accordingly.

For example, a high-bandwidth high-latency network would not be ideal for a request-reply type workload. It would be more suited to sending large batches of data before waiting for acknowledgement, perhaps looking at batch size and batch intervals to keep the batch open for longer. In addition, if work can be spread across multiple channels to exploit the high-bandwidth this may help.

On a low-bandwidth high-latency network, it might be appropriate to consider message compression, but there will be a trade-off with increased CPU costs which vary with compression algorithm. The usage of zEDC with channel compression type ZLIBFAST may reduce the compression costs.

Troubleshooting network issues

Some of our performance measurements started to show a regression in throughput when using TCP/IP to transport messages between 2 LPARs on the same sysplex. We had a good set of baseline data, so could see that the performance was significantly down, despite making no code changes nor applying any system updates. The tests used NETSTAT to monitor channel throughput and we could see occasional re-transmissions, particularly for larger messages (1MB). NETSTAT also showed the throughput rate on a "per second" basis for large messages was very varied.

The tests that were most impacted were streaming persistent messages from LPAR A to LPAR C. We tried moving the workload onto different LPARs e.g. moving from LPAR A to B and then LPAR B to C. The performance was only degraded when the target system was LPAR C.

This was an important factor in diagnosing the problem as we knew that LPAR A and B shared an OSA, and LPAR C had its own OSA. The connection from A and B went via a switch to C.

Changing the direction of the streaming so that the flow was LPAR C to A, and we got the expected (good) performance.

At this point we suspected a failing Small Form-Factor Pluggable (SFP) which converts optical connections to copper connections and are present in LAN/SAN cards/switches. We took the switch out of the equation by connecting the 2 OSAs directly - and the performance improved back to the expected baseline rate. We then tried putting the switch back in with a different cable, and the performance remained at the expected baseline rate, so the problem was cable-related.

What can we take from this?

- Know what your system does and the achieved throughput rates.
- Monitor your system, even when it is working well, so you have something to compare with, when an issue arises.
- If you need to make changes, do them one at a time so you can measure the impact of each specific change.
- Know what changes have been applied, so you can discount them if necessary.
- Sometimes it's not the software.

IBM MQ and zEnterprise Data Compression (zEDC)

IBM® zEnterprise® Data Compression (zEDC) capability and Peripheral Component Interconnect Express (PCIe or PCI Express) hardware adapter called *zEDC Express* were announced in July 2013 as enhancements to the IBM z/OS® V2.1 operating system and the IBM zEnterprise EC12 (zEC12) and the IBM zEnterprise BC12 (zBC12).

zEDC is optimised for use with large sequential files, and uses an industry-standard compression library. zEDC can help improve disk usage and optimize cross-platform exchange of data with minimal effect on processor usage.

There are a number of uses for zEDC which may affect the performance of the IBM MQ product including:

- Channel compression using ZLIBFAST. The performance benefits are discussed in [Channel compression on MQ for z/OS](#).
- [Compressing archive logs](#) to reduce storage occupancy.
- [IBM MQ and zEDC with SMF](#), which discusses the impact of capturing accounting data in a high volume, short-lived transaction environment.

On the IBM z15, the zEDC function was re-located from PCIe to on-chip, no longer being an optional feature. This re-location can result in improved performance of compression function. On-chip compression works in 2 modes.

1. Synchronous execution for problem state, as used by ZLIBFAST channel compression.
2. Asynchronous optimizations for large operations under z/OS, such as compressing archive logs.

Those operations performed using the synchronous execution mode see the largest improvement when moving to z15, with details available in the report [“MQ for z/OS on z15”](#).

Reducing storage occupancy with zEDC

Can I use zEDC with MQ data sets?

Sequential files allocated by using basic sequential access method (BSAM) or queue sequential access method (QSAM) can be compressed using the IBM zEnterprise Data Compression Express (zEDC Express) feature.

In an MQ subsystem, this means that archive data sets are eligible for compression.

What benefits might I see?

There are a number of benefits which using zEDC to compress MQ archive logs may bring:

- **Reduced storage occupancy of archive volumes**, meaning more archives can be stored on the same number of 3390 volumes. The compressibility of the messages logged will be the largest factor in the archive data set size reduction.
- **Reduced load on the IO subsystem**, which in a constrained subsystem could improve response rate on other volumes.

In our tests with dual logs and dual archives where the IO subsystems' cache was seeing increased disk fast write bypass (DFWBP) on the control unit used by both log copy 2 and the archive volumes, enabling archive log compression resulted in the response times from the I/O to log copy 2 reducing, with DFWBP being 0, which manifested in up to a 94% improvement in peak throughput with large messages.

What impact might I see?

The process of compressing, or indeed attempting to compress the MQ archive logs may result in a small increase in queue manager TCB costs. For a queue manager running a dedicated persistent workload with the intent to drive the MQ log process to its limit for a range of message sizes, we observed the queue manager TCB cost increase for the zEDC enabled measurements.

The following table is a guide to how much the queue manager TCB cost increased:

Message Size	4KB	32KB	1MB	4MB
Increase in QM TCB over non-zEDC measurement	+4%	+4%	+5-7%	+4-10%
Increase in peak throughput	0%	up to 4%	44-72%	50-94%

Note: Some of the increase in queue manager TCB is associated with the increased peak log rate, but there is some additional cost on MVS from the allocation/releasing of the target compression buffer plus some costs in setting up the call to the zEDC hardware. The increase in queue manager TCB is more significant with larger less compressible messages.

Reading the MQ archive data sets, such as when the “RECOVER CFSTRUCT” command was issued, was impacted when the archives were compressed using zEDC. This impact took the form of both a **reduced read rate** coupled with an increase in queue manager TCB costs for decompressing the data.

The following table summarises the results of a “RECOVER CFSTRUCT” command resulting in the recovery of 4GB of data.

	Uncompressed Archives	Archives compressed using zEDC
Recovery Rate (MB/sec)	60	18
Cost per MB (CPU ms)	1.7	2.4

The numbers in the table are based upon a MQ for z/OS 9.3 queue manager with 4GB of data stored on shared queues with data offloaded to SMDS. For the purposes of the backup and recovery tests, the queue manager is configured with single logs and single archives. The queue manager is configured with just 2 small active logs so that the majority of the data being backed up and recovered is stored on archive logs. The data on the queues is highly compressible. The costs are based on the measurements running on an LPAR with 3 dedicated processors of a 8561-7A1 (z15).

Note that when data is recovered solely from active logs, the recovery rate on our system was 370MB/second compared to the 60MB/second achieved when reading data back from archive logs.

When MQ is writing archive logs at a high rate the RMF PCIE report indicated that the single configured zEDC processor for the logical partition was running up to 70% utilised when compressing the dual archive logs for a single MQ queue manager. This peak usage occurred when the message was incompressible. With highly compressible messages at the peak logging rate, the zEDC processor was 50% utilised.

The PCIe I/O drawer in which the zEDC Express feature can be installed, can support up to 8 features with each feature able to be shared across 16 logical partitions. Sufficient zEDC features should be available to avoid impacting other users of the feature.

How we set up for testing

For the initial configuration we used Redbook “[Reduce Storage Occupancy and Increase Operations Efficiency with IBM zEnterprise Data Compression](#)”. Within this document, sections 4.2 to 4.5 were of particular interest as they discuss DB2 log archive data sets.

For our testing purposes, we already had a storage group for the MQ archive data sets, so we defined a dataclass of ZEDC, specifying ZR (zEDC required) for compaction and the data set name type of EXTENDED.

We also defined a storage group MQZEDC that was based on the existing MQMARCH storage group and added a similar number of volumes to the group.

Note, the zEDC feature was already enabled on the test system.

What to watch out for

The initial set up did not specify a value of EXTENDED for the data set name type - as a result measurements showed similar size archive and log data sets - indicating the data was incompressible or that no compression was attempted.

A subsequent review of the PCIE XML report produced by program ERBRMFPP indicated the zEDC processor was not being used.

The PCIE report can be generated by specifying “REPORTS(PCIE)” and viewing the contents of the XRPTS DD card, which contains data generated in XML format.

Note that on z15, the PCIE report has been replaced with the Extended Asynchronous Data Mover (EADM) report which can be generated by specifying “REPORTS(EADM)”.

Measurements

On our performance system, measurements were run using a range of message sizes from 2KB to 4MB.

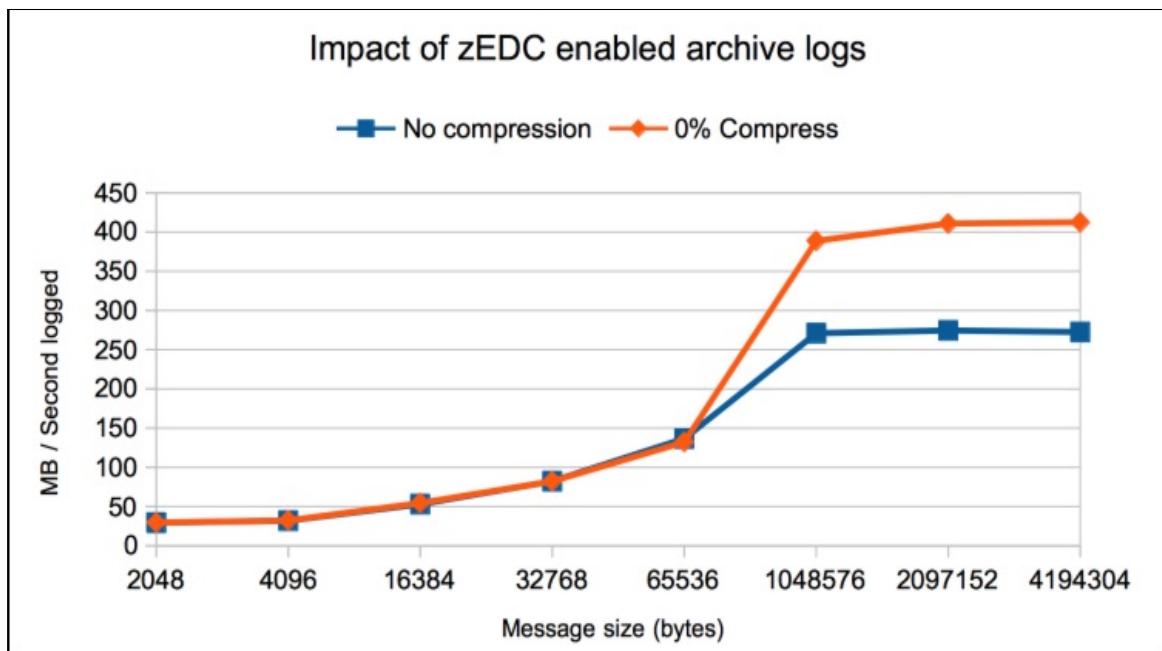
The queue manager was configured with dual logs and dual archives. There were sufficient logs available such that a slow archive would not delay log reuse.

Measurements were run on an LPAR with 3 dedicated processors on a 3906-7E1 with a dedicated DS8870 with FICON Express 16S links.

The I/O subsystem was configured such that log copy 1 used control unit 7004, and log copy 2 plus the archive logs used control unit 7128.

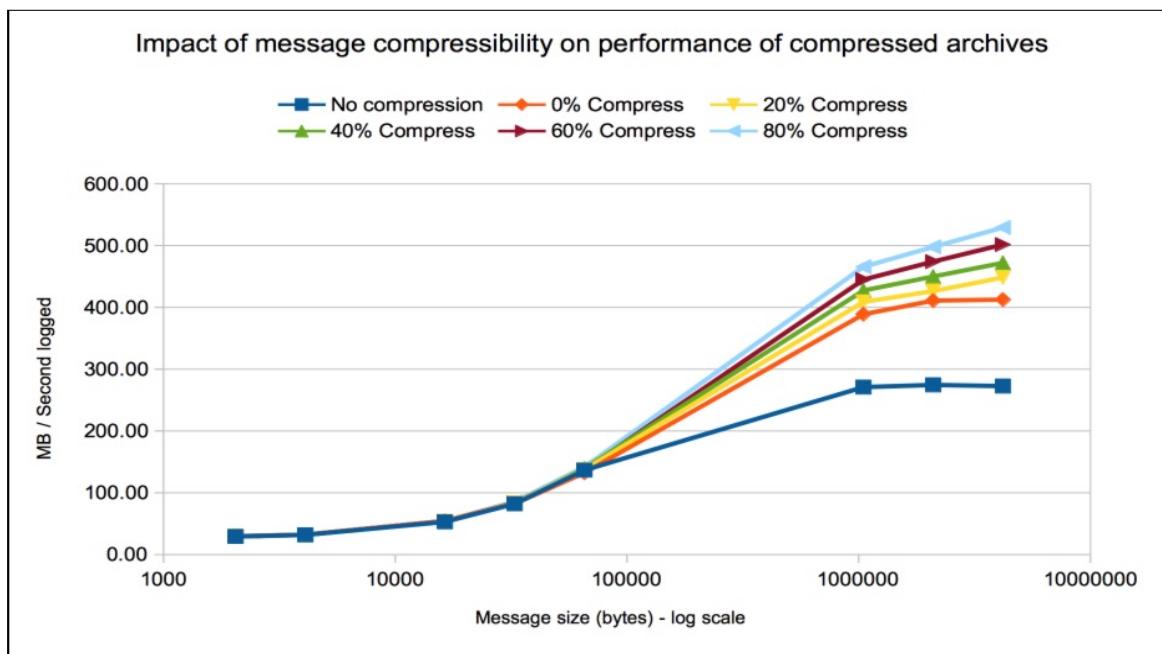
The following chart demonstrate the maximum sustained log rate where the data is minimally compressible.

In these measurements, an increase in log rate was observed when archive log compression was available for messages larger than 64KB, with a **50% improvement** in log rate with 4MB messages. *Given the data is uncompressible, this may be a side effect of moving to the EXTENDED archive data sets. Extended archive data sets have been supported since MQ version 8.0.*



How compressible is the data?

The following chart demonstrates the increase in log rate as the data becomes more compressible:



Highly compressible large messages benefit more from being compressed, and in our measurements where 4MB messages could be compressed by 80%, the **log rate increased by 94%**, to 529MB/second, compared to non-compressed archives.

Whether the I/O subsystem is constrained

As the data logged becomes more compressible, the amount of data written reduces, putting less load on the I/O subsystem.

With our configuration of the archive data sets using the same control unit as log copy 2, the MQ SMF data showed:

		CIs, Average I/O , After I/O , pages/I/O time in uSec, time in uSec,					
Log 1, 1 page	4273,	4273,	192,	5,	1		
Log 1,>1 page	43776,	4520380,	766,	2,	1e+02		
Log 2, 1 page	4273,	4273,	243,	20,	1		
Log 2,>1 page	43776,	4520380,	1218,	35,	1e+02		

Note the difference in average I/O time between log copy 1 and log copy 2, particularly when writing multiple pages.

We can also see from the Cache Subsystem Report that control unit 7128 shows signs of disk fast write bypass (DFWBP).

SSID	CU-ID	TYPE	CACHE	NVS	I/O	OFF	--CACHE HIT RATE-			-----DASD I/O RATE-----						
							RATE	RATE	READ	DFW	CFU	STAGE	DFWBP	IDL	BYP	OTHER
2100	7004	2107-961	507G	16G	7468	0.0	4296	3172	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2101	7128	2107-961	507G	16G	8995	0.0	520.9	8474	0.0	0.0	0.0	2727	0.0	0.0	0.0	0.0
2102	7407	2107-961	507G	16G	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

By contrast the equivalent measurement from the 0% compressible measurement shows no DFWBP on either control unit and the MQ SMF data shows:

		CIs, Average I/O , After I/O , pages/I/O time in uSec, time in uSec,					
Log 1, 1 page	5273,	5273,	201,	5,	1		
Log 1,>1 page	58617,	6057227,	811,	2,	1e+02		
Log 2, 1 page	5273,	5273,	183,	31,	1		
Log 2,>1 page	58617,	6057227,	897,	34,	1e+02		

Note the significant reduction in log copy 2's average I/O time, reducing from 1218 to 897 which is much more in-line with the log copy 1 time.

There is also a reduction in the DFWBP counts when compression is enabled as demonstrated in the following extract from the Cache Subsystem RMF report:

SSID	CU-ID	TYPE	CACHE	NVS	I/O	OFF	--CACHE HIT RATE-			-----DASD I/O RATE-----						
							RATE	RATE	READ	DFW	CFU	STAGE	DFWBP	IDL	BYP	OTHER
2100	7004	2107-961	507G	16G	9754	0.0	5607	4147	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2101	7128	2107-961	507G	16G	6701	0.0	695.4	6006	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2102	7407	2107-961	507G	16G	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

IBM MQ and zEnterprise Data Compression (zEDC) with SMF

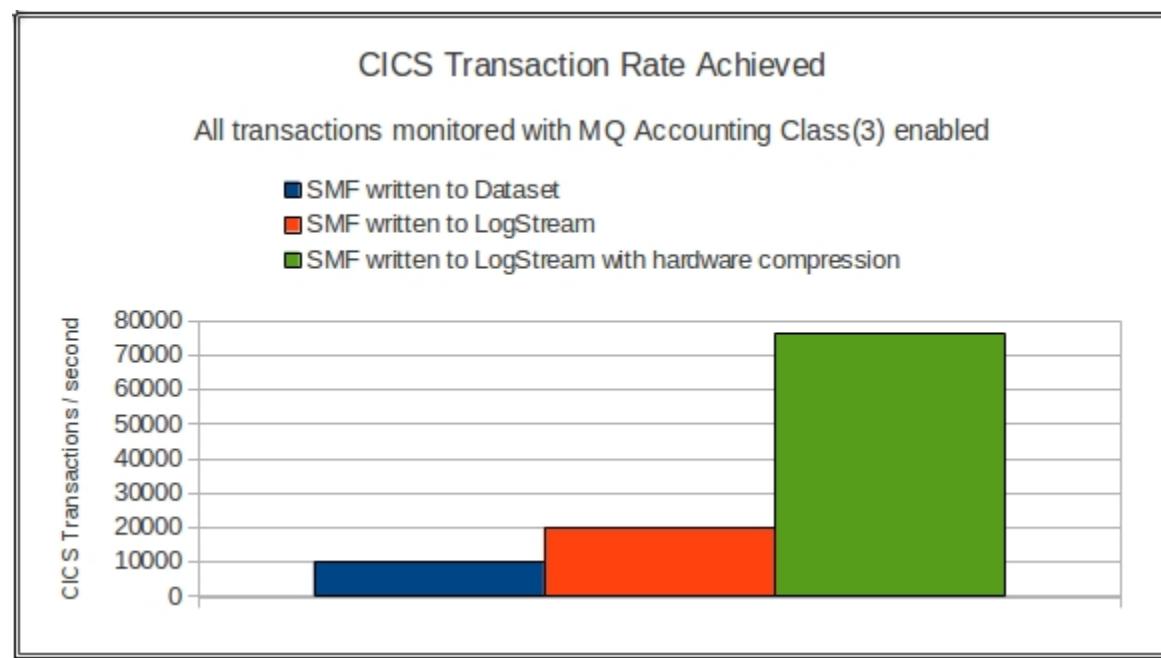
IBM MQ allows customers to enable accounting data to be captured and written to the MVS System Management Facility (SMF).

We have found that with short-lived transactions, such as CICS transactions, that the transaction rate can exceed the rate at which IBM MQ can write data to SMF. As a result, some accounting data can be lost – and this loss is reported via the MQ queue manager job log.

In z/OS v1.9, SMF allowed data to be written to log streams rather than data sets and this gave an immediate improvement to the rate at which IBM MQ can write accounting data to SMF.

Note: When the transaction rate exceeds the rate at which IBM MQ can write to logstreams, warning messages are only written to the system log.

Using z/OS 2.1 and the zEDC feature, log streams can be compressed using hardware compression which can significantly increase the rate at which MQ can sustain accounting data being written to SMF.



The diagram above shows that moving from SMF data sets to log streams resulted in a doubling of the transaction rate without loss of accounting data.

Using hardware compression on SMF log streams in conjunction with IBM MQ's accounting trace allowed a further 3.8 times higher throughput through a single queue manager to be recorded for accounting purposes.

Data set encryption

Data set encryption (DSE) was originally implemented in z/OS v2r2, and initial MQ support in version 8.0 was limited to archive logs and BSDS.

In MQ 9.2, support for data set encryption is enabled for active logs, page set and shared message data sets (SMDS).

Table: MQ data set type and encryption support statement

Data set	Pre-9.2	9.2
BSDS	Supported	Supported
Sequential	Supported	Supported
Page set 0	No - Abend 5C6-00C91400	Supported
Page sets 1-99	No MQRC 2193 "Pageset error"	Supported
Active logs	No - Abend 5C6-00E80084	Supported
Archive logs	Yes - V8 onwards	Supported
SMDS	No SMDS marked avail(error)	Supported

Notes on table:

- Data held in buffer pool and SMDS buffers is not encrypted. The use of appropriate AMS policies would ensure encryption of the data held in the buffer pool or SMDS.
- Prior to MQ 9.2, applying data set encryption to SMDS could result in unexpected behaviour depending on the offload rules and potentially how full the structure gets. For example if the SMDS status “avail(error)” is not noticed at the time the message is logged, an application error may not occur until the MQPUT of a message that requires offload is attempted. If the application only uses small messages, that offload may not occur until the CF structure is 80% full, which could be a significant time between the SMDS error being logged and the MQPUTs failing as a result.

Why use data set encryption

z/OS data set encryption provides enhanced data protection for z/OS data sets, giving clients the ability to encrypt all of the data associated with entire applications and databases, without the need to make application changes and without impacting SLAs.

Additional design advantages provided by z/OS data set encryption are:

- Uses CPACF acceleration in collaboration with Crypto Express for protected key cryptography, enabling cryptographic operations to be hardware accelerated while ensuring that key material is not visible in the clear to the OS, Hypervisor or application.
- Encrypt data by policy in a way that is aligned with clients' current access control mechanisms, offering a simplified configuration experience.
- Encrypts at-rest data in bulk, performing efficiently at speed and for low-cost.
- Allows data to remain encrypted with keys managed on IBM Z during replication, backup, and migration.
- Can be configured such that encryption keys are owned and managed by a logical organisational environment, providing cryptographic separation between environments.
- Reduce the risks associated with undiscovered or mis-classified sensitive data.
- Encrypting all of the data associated with an application or database can simplify and reduce the cost of compliance.

The data set encryption feature allows AES encryption of data contained in named data sets using ICSF and RACF, so preventing visibility of sensitive data from systems administrators with a legitimate need to manage those data sets.

Data set encryption with the MQ queue manager

Encryption of data is not free, and as such, consideration should be given as to whether your system has sufficient capacity to support encrypted MQ data sets without impacting the performance of your workloads.

The following sections discuss the additional cost from encrypting MQ data sets based on observations on our dedicated performance systems. To try to simplify the impact, the costs are discussed in three configurations:

1. [Active and archive log data sets.](#)
2. [Page sets.](#)
3. [Shared message data sets.](#)

Typically we might expect that if you are planning on encrypting any of the MQ data sets, you would encrypt **all** MQ data sets, i.e. page set, active and archive logs, SMDS and BSDS.

BSDS might be regarded as less important to encrypt as it contains systems rather than application data, but this may depend on your sites encryption policy.

Despite MQ archive logs supporting data set encryption since IBM MQ for z/OS version 8.0, active and archive logging are closely coupled, and so we offer a section on encrypting all of your MQ log data sets.

The I/O to MQ page set is somewhat different and will depend upon the nature of the workloads using the queue manager. As such, the section on encrypting page set is kept separately from MQ logs and SMDS.

Active and Archive log encryption

Before discussing the impact of encrypting the MQ active and archive logs, consider what happens when logging occurs where the MQ queue manager has dual active and dual archive logs, all of which are encrypted.

- Persistent message workload causes MQ queue manager to **encrypt+write** to active log copy 1 and **encrypt+write** to active log copy 2 data sets.
- At end of current active log:
 - Queue manager reports CSQJ002I “END OF ACTIVE LOG DATA SET”.
 - Current active log(s) switches to next active log(s).
 - Checkpoint starts for all buffer pools.
 - Archive task started, processing the recently filled active log:
 - Select either log copy 1 or 2 to provide the data to archive.
 - **Read+Decrypt** chosen active log copy - 1 page at a time.
 - **Encrypt+write** to active log copy 1.
 - **Encrypt+write** to active log copy 2.

The point of this is to show that dual logging and/or dual archiving causes multiple encrypt calls as well as decrypting the data read from the active logs.

With dual logging and dual archiving all protected with data set encryption, the queue manager is encrypting the data 4 times, plus decrypting the data once, for a total of 5 encrypt/decrypt calls for each page logged.

Table: Cost of data set encryption on MQ active and archive logs

Log type	CPU microseconds per 4KB page	CPU microseconds per MB
Archive logs	+0.6	+154
Active logs	+0.7	+180

Notes on table:

- Active logs are accessed via Media Manager, which has additional overheads.
- Costs shown are based on the average across a range of message sizes:
 - Archive logs: range between 0.25 to 1 microseconds per 4KB page.
 - Active logs: range between 0.6 to 0.9 microseconds per 4KB page.

Since we know the impact of data set encryption to both active and archive logging, we can apply this to our queue manager using the following example.

Example: MQ configured with dual active and archive logs, with 1GB log data sets.

Log type	Cost / MB	MB in active log	Factor	Cost (Cost * MB * Factor)
Archive logs	154	1024	2	315 CPU milliseconds
Active logs	180	1024	3	550 CPU milliseconds

Notes on table:

- Factor of 2 used for the separate **encrypt** and write to each copy of the archive log.
- Factor of 3 used for the separate **encrypt** and write to each copy of the active log plus the read and **decrypt** of the log for the archiving task.

In this example, for each 1GB of MQ workload logged, there would be approximately 0.86 CPU seconds *additional* cost in the MQ MSTR address space as a result of enabling data set encryption for both active and archive logs.

Page set encryption

The cost of encrypting MQ page sets is slightly more complicated than encrypting MQ logs due to types of I/O performed.

The program MQSMF, available in supportPac MP1B “Interpreting accounting and statistics data”, provides a number of page set related reports, but the one of interest here is PSET. Note that only selected data is shown in the following sample:

```
PS01 BP 1, Pages 1040334, Size 4063MB, ... Pageset is encrypted
      Pages written in checkpoint      2918219
      Pages written not in checkpoint  10289

PS01 Type :I/O requests.    Pages, Avg I/O time, pages per I/O
PS01 Write:     182403,      2918331,        1968,       16.0
PS01 IMW :      10177,       10177,         274,        1.0
PS01 GET :      1,           1,            378,        1.0
```

As highlighted in the sample report, there are three types of I/O performed on MQ page sets, namely WRITE, IMW (immediate write) and GET:

- **GETs** are performed when MQ determines it necessary to read (and decrypt) from page set. Reads are performed one page per I/O request.
- **IMWs** (Immediate writes) occur when the buffer pool reaches 95% full and the data is moved from buffer pool to page set synchronously. Immediate writes are performed 1 page per I/O request. This is an indication that the buffer pool is not sufficiently large for optimal performance.
- **Writes** are performed at 2 points - at checkpoint and when the buffer pool reaches 85% full (also termed “written not in checkpoint”). In each instance, each I/O request will write up to 16 pages.

In terms of the additional cost of data set encryption on page set I/O:

- **GET** +0.7 microseconds / page read.
- **IMW** +0.7 microseconds / page read.
- **WRITE** +2.6 microseconds / page (+41.6 microseconds per I/O).

We can use these numbers in conjunction with the output from the PSET report to estimate the cost impact from applying data set encryption to MQ page sets e.g.

- **GET**: 1 request for an additional 0.7 microseconds, i.e. not a discernible impact from page set encryption.
- **IMW**: 10,177 requests for 10,177 pages - an additional cost of 7124 CPU microseconds.
- **WRITE**: 182,403 requests for 2,918,219 pages - which we would predict at costing an additional 7.59 CPU seconds.

Shared message data set encryption

Typically for messages that can be stored in their entirety in the Coupling Facility, there is no significant impact to the transaction cost from encrypting the SMDS. This is true even when the CF structure is encrypted.

Due to the way that SMDS works, the majority of the encryption costs will be incurred by the application performing the MQPUT, rather than the queue manager address space.

Decrypting the data held on SMDS is charged to the queue manager address space and can partially be offset by reduced cost in the application issuing the MQGET. This is discussed further in “[Why are unencrypted gets more expensive than encrypted?](#)”

Furthermore, when accessing messages stored in the local queue manager’s SMDS, sufficient DSBUFS can mean the message can be accessed from buffer rather than needing to read and decrypt the data physically stored on the data set.

When data is read from a remote queue managers’ SMDS, the decryption costs are charged to the local queue manager performing the read of the SMDS.

Message persistence does not impact the cost of encryption or decryption of SMDS data.

MQPUT to SMDS

The cost of an MQPUT of a message to a shared queue where the message is offloaded to SMDS is largely attributed to the application address space. For non-persistent messages the cost in the queue manager address space is minimal compared to the application cost.

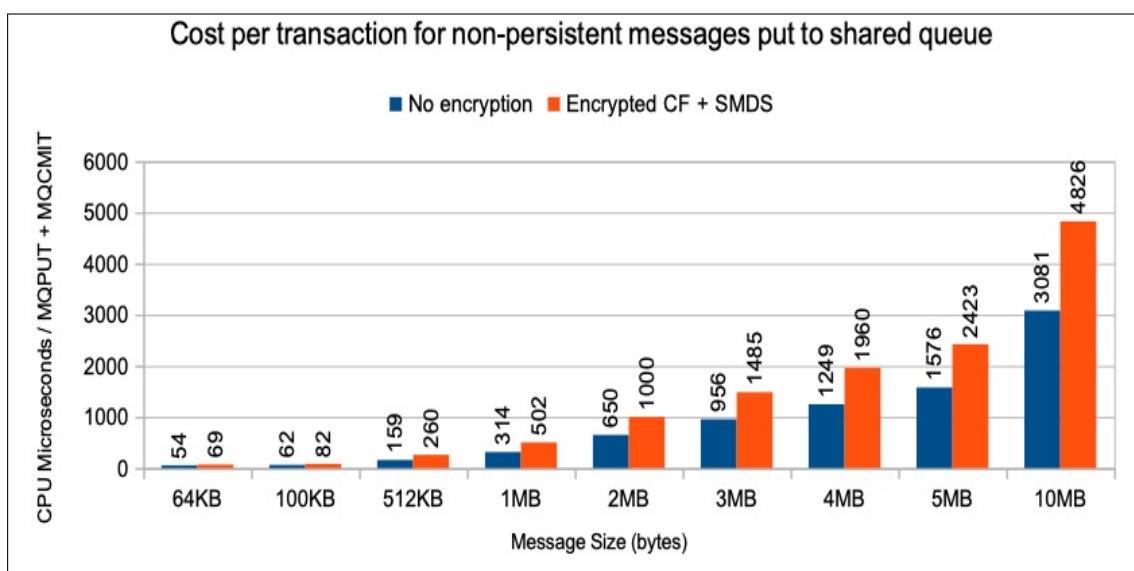
As a result of encrypting the SMDS, putting messages to shared queues where the message is written to SMDS does result in increased cost to the putting application - whether a local application or the channel initiator if the message arrives via an MQ channel.

The following chart compares the cost of non-persistent messages put to a shared queue for 2 configurations:

1. Neither CF nor SMDS are encrypted.
2. Both CF and SMDS are encrypted.

The application performing the MQPUT generates the message once, and performs multiple iterations of MQPUT and MQCMIT before ending. There is no business logic in the program so the costs reported are largely equivalent to the data reported by MQ Accounting class(3).

Chart: Compare cost of MQPUT to Shared Message Data Set



Note: The increase in cost from encrypting the SMDS equates to approximately 0.7 CPU microseconds per 4KB page on IBM z15 regardless of message persistence.

MQGET - When the messages are read from SMDS buffers

As mentioned earlier, when sufficient DSBUFS are available that the messages can be gotten from SMDS buffers, the cost of the MQGET remains the same regardless of whether the data set is encrypted.

Note: Configuring more DSBUFS to minimize the effect of decrypting data can have a negative effect - due to the time taken to locate the next available buffer (oldest). For example puts and gets to structures defined with DSBUF(3000) were significantly more expensive than messages put to structures defined with DSBUF(10-200).

MQGET - When the messages are read from local SMDS

The increase in cost from decrypting the message on the SMDS is approximately 0.7 CPU microseconds per 4KB page, regardless of message persistence.

In the queue manager address space there was an increase of approximately 0.8 CPU microseconds per 4KB page in SRB usage. However some of this cost increase was offset by a reduction in the application address space equivalent of 0.1 CPU microseconds per 4KB page. This is described in further detail in [Why are unencrypted gets more expensive than encrypted?](#)

MQGET - When the messages are read from remote SMDS

The increase in cost from decrypting the message on the SMDS is approximately 0.8 CPU microseconds regardless of persistence.

In the queue manager address space there was an increase of approximately 0.9 CPU microseconds per 4KB page - as with reads from local encrypted SMDS'. However, as with reads from the local SMDS, some of this cost increase was offset by a reduction in the application address space equivalent of 0.1 CPU microseconds per 4KB page.

When reading from a remote queue managers' SMDS, there are slightly reduced costs charged to the local queue manager, but this is offset by a small increase in the CPU used by the remote queue manager.

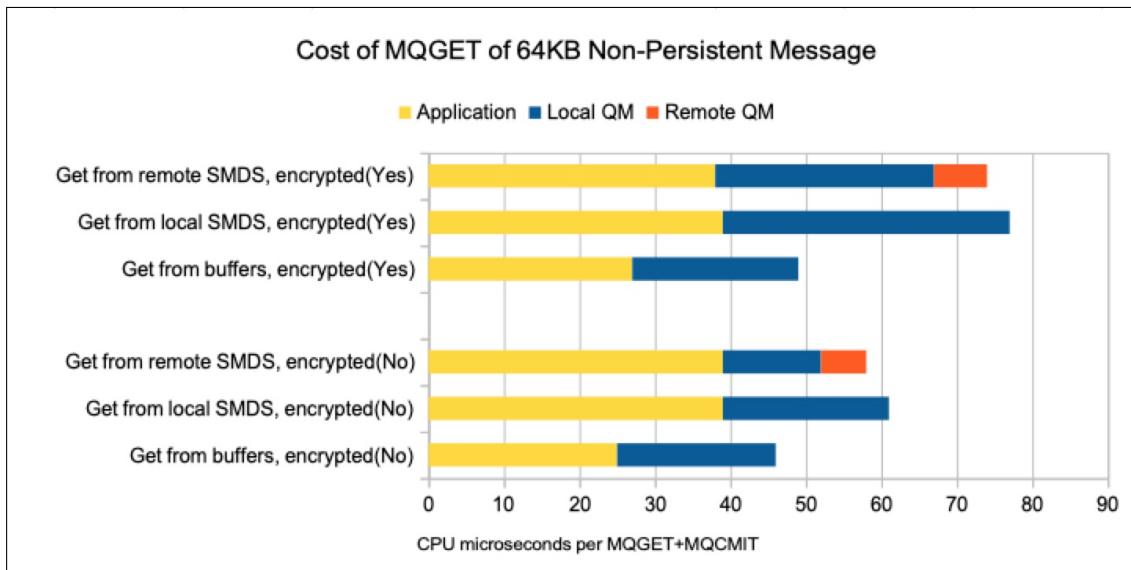
This CPU usage in the remote queue manager usage is due to the remote queue manager deleting the message from its own SMDS and is not affected by data set encryption status.

Our measurements suggest the total cost of MQGET from a remote queue managers' SMDS is within 5% of the cost of an MQGET from a local queue managers' SMDS.

Comparing the cost of MQGETs from shared queue

The following charts offer comparisons of MQGET costs, by address space, for the configurations of MQGETs described previously, for three message sizes - 64KB, 1MB and 10MB.

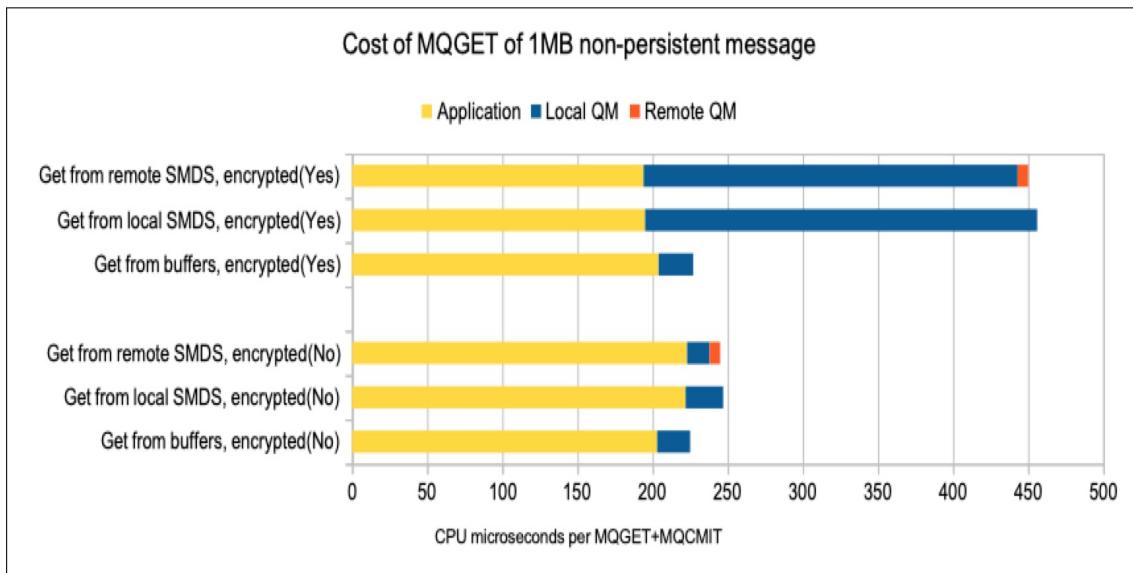
Chart: Compare cost of MQGET of 64KB non-persistent shared queue message



Notes on chart:

- The cost of decrypting the message held on SMDS is charged to the queue manager address space.
- The total cost of MQGET from a remote SMDS is similar to the cost of an MQGET from a local SMDS.
- When a message can be got from SMDS buffers, the encryption status of the SMDS does not affect the cost of the MQGET.
- The cost to the application address space of the MQGET is slightly less when accessing encrypted data than when accessing unencrypted data.

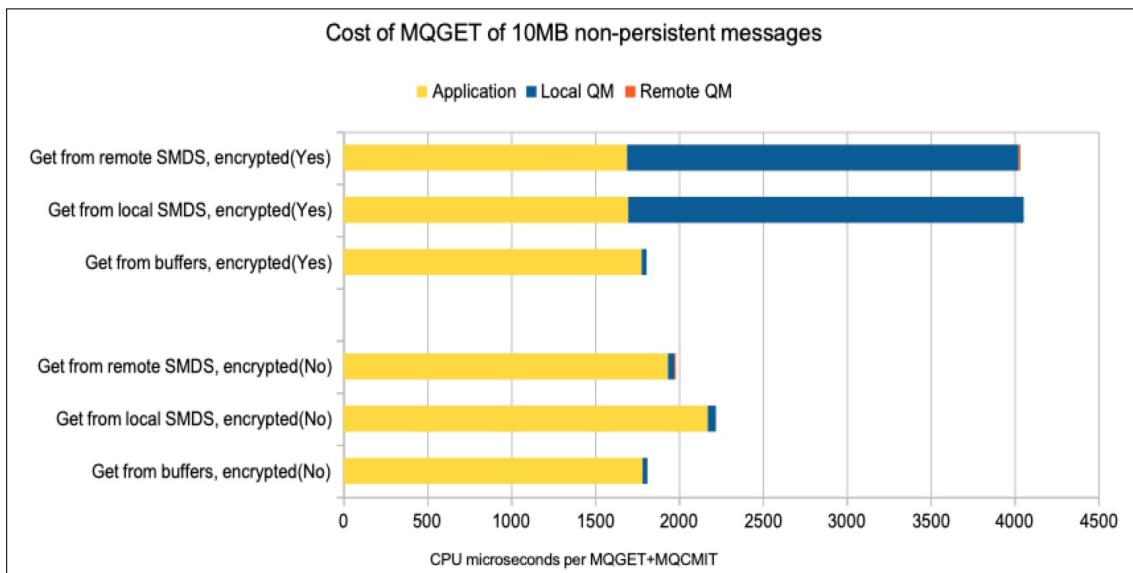
Chart: Compare cost of MQGET of 1MB non-persistent shared queue message



Notes on chart:

- The total cost of the MQGET from a remote SMDS is similar to the cost of the MQGET from a local SMDS - within 5%.
- With these 1MB messages, the application cost, i.e. the reported cost of the MQGET as per MQ accounting class(3), is 13% lower when accessing encrypted data.

Chart: Compare cost of MQGET of 10MB non-persistent shared queue message



Notes on chart:

- With these 10MB messages, the application cost, i.e. the reported cost of the MQGET as per MQ accounting class(3), is 22% lower when accessing encrypted data. This does not take into account the increase in queue manager address space costs from decrypting the data.

Why are unencrypted gets more expensive than encrypted?

According to the MQ Accounting class(3) data and as can be seen in the “[MQGET of 1MB](#)” and “[MQGET of 10MB](#)” message charts, it appears that the MQGET costs are lower when getting data from encrypted SMDS than when accessing data from unencrypted SMDS.

For example:

Table: Accounting class(3) data for MQGET of 10MB messages

	Unencrypted	Encrypted
Average CPU time	1923 CPU microseconds	1683 CPU microseconds

In this example, the savings from encryption equate to approximately 0.1 CPU microseconds per 4KB page.

This does not take into account the increased cost in the queue manager address space from decrypting the message.

The increase in queue manager cost from MQGETs of message in encrypted SMDS is in SRB and there are no MQ-based statistics to report details of this cost.

Summary of data set encryption costs with the MQ queue manager

As has been discussed in this data set encryption section, the cost of encrypting MQ data sets does not come at zero cost.

The following provides a summary of the CPU overhead from encrypting each of the types of MQ data set, but it is worth considering the impact as a whole, i.e. persistent shared queue messages will see the costs rise due to encrypted active logs, encrypted archive logs as well as encrypted shared message data sets.

Active and archive log data sets costs when operating with data set encryption can be summarised as follows:

- Archive logs cost increases with data set encryption by approximately 0.6 CPU microseconds per 4KB page written.
- Active logs cost increases with data set encryption by approximately 0.7 CPU microseconds per 4KB page read / written.
- Remember to factor in whether running with single or dual logging / archiving.
- Remember that when using encrypted active logs, the data must be decrypted prior to archiving - even if the archives are encrypted too.

Page set costs when operating an encrypted data set can be summarised as follows:

- **GET** +0.7 microseconds / page read.
- **IMW** +0.7 microseconds / page read.
- **WRITE** +2.6 microseconds / page (+41.6 microseconds per I/O).

Shared message data set costs when operating an encrypted data set can be summarised as follows:

- Increased MQPUT costs of approximately 0.7 CPU microseconds per 4KB page.
- Increased MQGET costs of approximately 0.7 CPU microsecond per 4KB page when the data is read from the local queue managers shared message data set.
- Increased MQGET costs of approximately 0.8 CPU microsecond per 4KB page when the data is read from a remote queue managers shared message data set.

The impact of encrypted SMDS can be mitigated for MQGETs when the queues are sufficiently low in depth such that messages can be gotten from SMDS' buffers. Note that it is not recommended to significantly increase the number of SMDS buffers to avoid gets from encrypted SMDS as this can increase the cost of both MQPUTs and MQGETs.

Final observations on data set encryption with MQ:

- The cost per page when writing multiple pages per I/O request is higher than the cost of a single page I/O request.
- Costs may vary depending on how busy your system is - lightly loaded systems or tasks, such as the MQ logger, may see a higher cost per page than reported in this section.
- Data set encryption costs were generally similar on z14 and z15 for our measurements, as they used AES encryption which has been optimised on CPACF. Costs may be higher on earlier generations of IBM z hardware.

- Consider that in CPU constrained systems that the additional cost from encryption may impact existing workload characteristics, resulting in more data written at checkpoint or needing to be read from page set.
- Page set immediate writes and gets can be minimised with sufficiently large buffer pools.
Over-sized buffer pools can minimise immediate writes to page set.
 - Over-sized buffer pools are buffer pools that are sized at 105% of the corresponding page set.
 - By setting the buffer pool to 105% of the size of the page set, you can avoid additional I/O's for immediate writes.
 - When multiple page sets map to a single buffer pool, over-sizing the buffer pool may not have the desired impact of avoiding immediate writes.

zHyperWrite support for active logs

MQ 9.2 introduces support for zHyperWrite with MQ's active log data sets.

This section discusses improved active log performance, in particular with IBM® Metro Mirror and zHyperWrite.

IBM® Metro Mirror, previously known as Synchronous Peer to Peer Remote Copy (PPRC), is a synchronous replication solution between two storage subsystems, where write operations are completed on both primary and secondary volumes before the write operation is considered to be complete.

The IBM MQ Knowledge Centre section on "[Using MetroMirror with IBM MQ](#)" discusses which types of IBM MQ data sets can be replicated using Metro Mirror.

As of MQ 9.2, zHyperWrite is supported with MQ active logs. This offers benefits in a number of areas:

- Reduced I/O times by **up to 60%**.
- Reduced elapsed time for MQ commit by **up to 60%**, which can reduce contention.
- Reduced elapsed time for MQ put of persistent messages by **up to 33%**.
- Improved sustained log rate, allowing for each MQ queue manager to process **up to 2.4 times** the volume of workload.

Caveats:

- The benefits of zHyperWrite reduces as the distance of replication increases.
- Whilst improved log performance can be achieved with zHyperWrite, increased SRB time in the MQ MSTR address space might be observed because the extra (Media Manager) I/O requests are processed under MSTR SRB when zHyperWrite is enabled.
- It can be beneficial to configure additional channel paths (CHPIIDs) from z/OS specifically for the zHyperWrite secondary volumes.

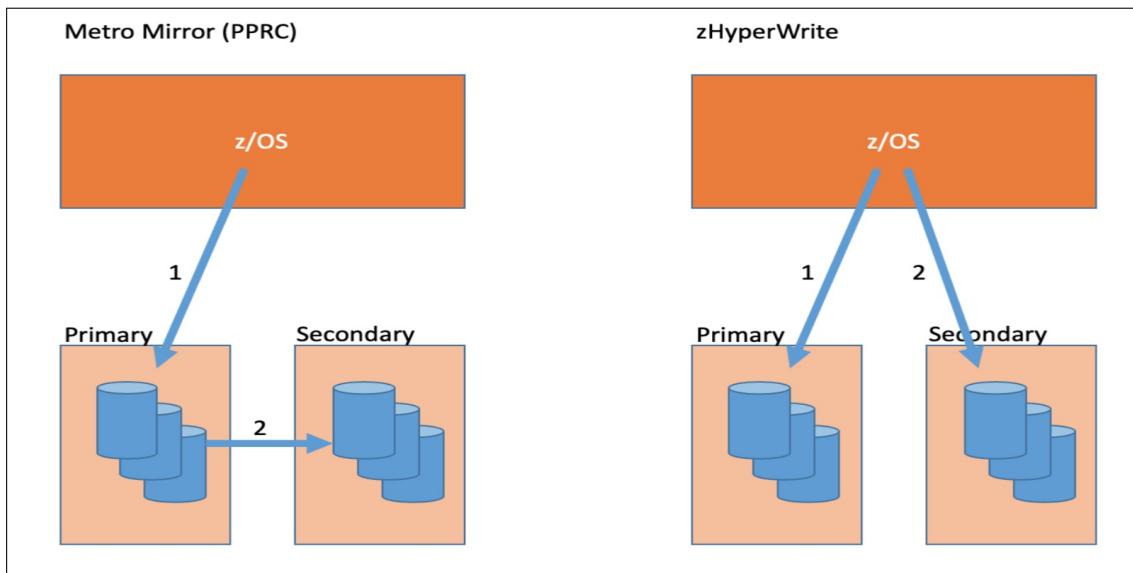
What is zHyperWrite?

The z/OS 2.4 Knowledge Centre describes zHyperWrite as:

IBM zHyperWrite processing can be used by I/O drivers, such as Media Manager, for certain write I/O operations to perform software mirroring to peer-to-peer remote copy (PPRC) devices that are monitored for HyperSwap® processing (with GDPS® or TPC-R). IBM zHyperWrite data replication can be used to reduce latency in these HyperSwap environments. For maximum benefit, IBM zHyperWrite data replication should only be used when all synchronously mirrored relationships are managed by HyperSwap. Devices support IBM zHyperWrite data replication when the following conditions are true:

- The devices support IBM zHyperWrite data replication. Both the primary and secondary devices in a synchronous PPRC relationship must support this function.
- The devices in the synchronous PPRC relationship are managed by HyperSwap (either GDPS HyperSwap or TPC-R HyperSwap).

A simplified view of the difference between PPRC and zHyperWrite is offered in the following diagram:



Notes:

In the PPRC configuration, the mirroring is driven from the primary control unit.

In the zHyperWrite configuration, the mirroring is requested once the I/O to the primary control unit is requested. This can reduce the time to initiate the mirrored request.

Why does my log performance matter?

Improved MQ active log performance is important for a number of reasons:

- Allows more work to get done with the same hardware footprint.
- Better able to handle workload spikes.
- Reduces cost.

Generally, there are a number of ways to make transactions run faster on System Z and z/OS:

1. Faster CPU.
2. Software scaling, reducing contention, faster I/O.
3. Faster I/O technologies such as zHPF, 16Gb channels, zHyperWrite, faster DASD with larger cache sizes, etc.
4. Run at lower utilisations, address Dispatcher Queuing Delays.
5. Faster network, such as SMC-R or SMC-D.

When using high levels of persistent messaging, the rate at which MQ is able to process the workload is largely dependent on the rate at which the MQ logger task is able to complete its I/O. This rate is almost entirely dependent upon the rate at which the I/O subsystem is able to respond to the requests.

The following extract from MQ Statistics data, formatted using program MQSMF (part of support-pac [MP1B](#) “Interpreting accounting and statistics data”), demonstrates the single logger task, which runs as an SRB, is 99.82% busy, of which 95.90% is waiting for I/O to complete:

Log write rate	272MB/s per copy
Logger I/O busy:	95.90%
Logger task busy:	99.82%

In this environment, one way to alleviate this constraint is to reduce the I/O times.

When running synchronous replication technologies such as Metro Mirror (PPRC), the I/O times can be significantly higher than when running without the datasets replicated. The zHyperWrite technology can significantly reduce the I/O times when replicating datasets.

There are a number of benefits that zHyperWrite-enabled active logs can offer to a MQ queue manager:

1. [Reduced I/O time](#).
2. [Reduced elapsed time for MQ commit and MQPUT](#).
3. [Improved sustainable log rate](#).

These benefits do not come at zero cost, and indeed there is an impact to the queue manager costs, which is discussed in the “[Impact to MQ queue manager costs](#)” section.

zHyperWrite test configuration

The following measurements were run on a single LPAR of the 3 LPAR MQ Performance sysplex on the IBM z16 (3931-7x1). In these tests, the LPAR was configured with 3 dedicated general purpose processors, and for the purposes of comparison is similar to a 3931-703, connecting to IBM DS8950F DASD.

A single MQ queue manager was created in 3 configurations, in each case with dual active logs and dual archive logs.

1. Baseline - no mirrored datasets.
2. PPRC - active logs mirrored using Metro Mirror.
3. zHyperWrite - active logs mirrored using Metro Mirror and zHyperWrite enabled at the queue manager.

Previously, measurements on IBM z15 using DS8870 DASD had the queue manager configured with single active and archive logs. In the dual active and archive log configuration we saw measurements impacted by site hardware limitations. Details of the observed impact can be found in the “[Impact of I/O limitations on dual active and dual archive logs on older hardware](#)” section of the report.

With the updated DASD the limitations previously observed were alleviated and for these measurements, the queue manager was configured with dual active and dual archive logs.

In each configuration, a put-commit/get-commit workload was run using a range of message sizes from 2KB to 4MB.

In the test configuration used, the mirrored DASD was at short distance (less than 1KM). Synchronously replicated DASD over extended distance may see different results.

With regards to the disk subsystem, the tests were run on a single frame DS8950F dedicated for performance testing, with dedicated links between the z16 and the DS8950F.

The PPRC configuration has 4 dedicated 16Gb Fibre channels for the PPRC-specific traffic that is routed from one half of the DASD via a switch to the other half.

I/O driven directly from z/OS is spread across 4 dedicated 32Gb Fibre channels.

To avoid contention from the zHyperWrite mirror write requests, 2 of the 4 channel paths were configured for the I/O to the secondary (mirror) volumes and all other I/O was routed through the remaining 2 channel paths.

Ideally there would have been 4 channel paths for the primary writes, which included I/O to both active logs and archive logs and all MQ page set I/O, with 2 additional channel paths for the secondary writes to the mirror of the active logs.

In all configurations, zHPF (z High Performance FICON) was used.

The measurements ran for a number of SMF intervals and were designed to run the MQ logger task at the maximum sustainable throughput rate.

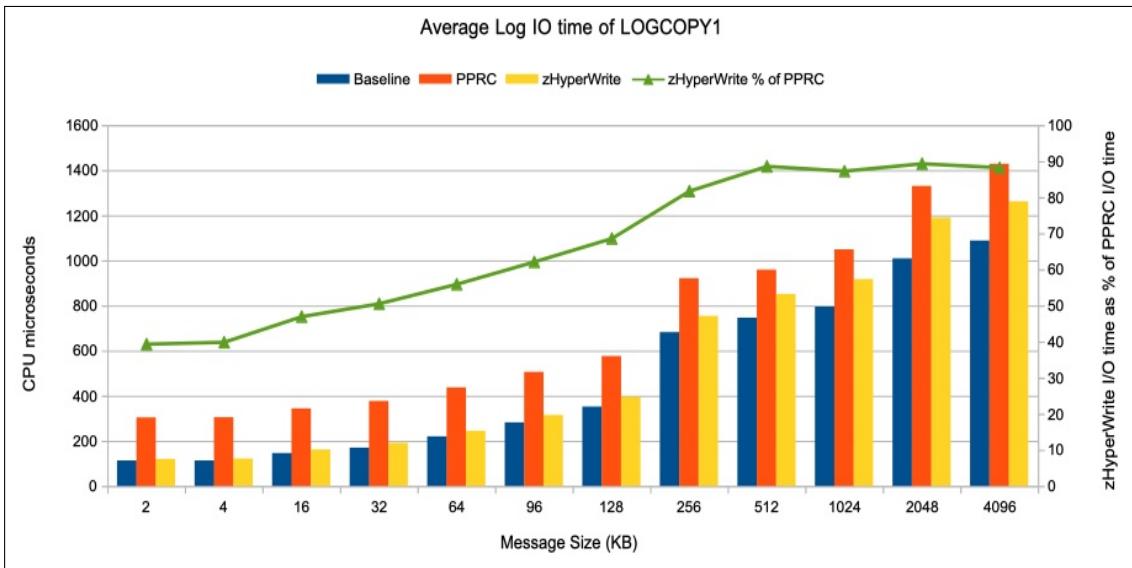
Reduced I/O time

When a persistent workload is throughput constrained, it can be due to the MQ logger task running at capacity. The overall throughput may vary depending on message size(s), and to achieve a higher throughput there are several options:

- Split the workload over multiple queue managers.
- Reduce the I/O response time.

The following chart uses the MQ Statistics data to plot the average I/O times for the logger task for each of the three specified configurations:

Chart: Average I/O time for write to LOGCOPY1



For messages up to 96KB, zHyperWrite I/O is taking less than 60% of the time of the PPRC I/O requests.

Once the messages exceed 96KB, the I/O times get closer, but in this configuration where the I/O subsystem is not constrained, zHyperWrite shows I/O times at least 10% less for all message sizes when compared with the PPRC configuration.

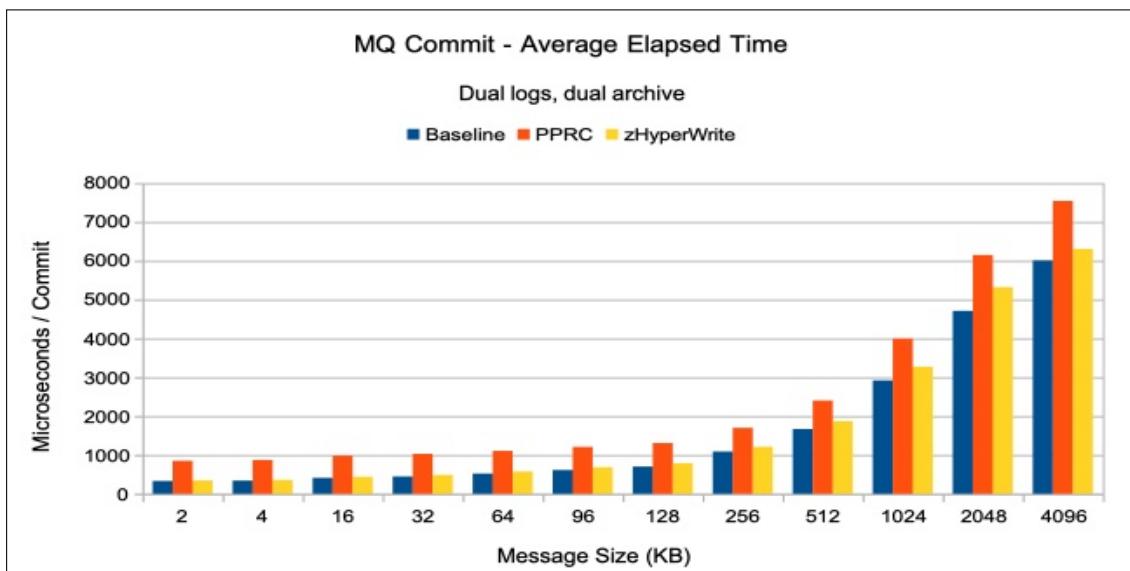
Reduced elapsed time for MQ commit

In transactions, the wait time for the MQ log write I/O, typically at commit time, often makes up a significant proportion of the transaction latency, particularly when disk replication is enabled.

zHyperWrite is designed to reduce the latency of the log writes in a synchronous peer-to-peer remote copy (PPRC) environment. With zHyperWrite enabled, MQ triggers a log write I/O to the secondary disk subsystem in parallel to the log write I/O to the primary disk subsystem. By overlapping the two I/Os, which are run serially without zHyperWrite enabled, a significant reduction in MQ log write I/O write time is possible.

The following chart shows the average elapsed time of the MQ commit for a range of message sizes.

Chart: Average elapsed time of MQ commit



The reduction in average commit time ranges from 15% to 60% less than the PPRC configuration.

In some cases, another indirect benefit of zHyperWrite can be observed. The reduced log write wait times as a result of having zHyperWrite enabled, can lead to reductions in other types of contention, which in turn can result in more CPU time savings because MQ has fewer contentions, both suspend and resume, to manage.

Note that for most messages, the amount of data logged at put time far exceeds the amount of data logged at get time. See “[How much log space does my message use?](#)” for guidance on how much data is logged.

In our measurements the elapsed time of the MQPUT was reduced by up to 33% when comparing the zHyperWrite with PPRC (Metro Mirror) configuration.

Improved sustainable log rate

The chart in this section shows the rate in MB per second that the dual active log was able to sustain on our system.

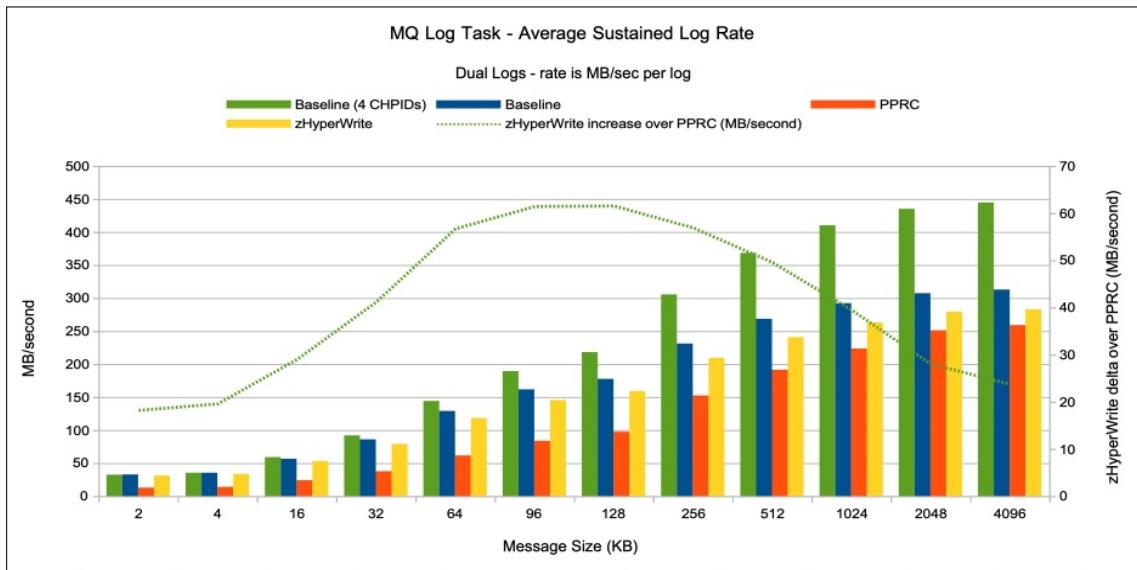
In all cases, the queue manager is configured with dual logs and dual archives, so for every MB put by the application, there are writes of between 4MB (for the baseline) to 6MB (for PPRC or zHyperWrite) in total volume to the disks i.e.:

- 2 MB to active log (1 MB per log)
- 2 MB to archives - once the test is running, the queue manager is driving the archive process constantly.
- 2 MB to the mirrored logs (1 MB per log).

What this means is that when the charts show a logging rate of 250 MB / second, there is approximately 1 GB / second written to the I/O subsystem in the baseline configuration and 1.5 GB / second in the PPRC and zHyperWrite configuration.

With the larger cache size on DS8950F, there was no indication of disk cache saturation that we saw previously on DS8870, which is discussed in the section “[Impact of I/O limitations on dual active and dual archive logs on older hardware](#)”.

Chart: MQ Log task - sustained log rate



Notes on chart:

The baseline performance has 2 sets of data, one where all four CHPIDs were available for both active logging and archiving, and the second where only 2 CHPIDs were available. With larger message sizes, limiting the baseline measurement to two CHPIDs reduces the capacity by up to 30%.

The green line indicates the difference in MB/second that zHyperWrite offers over PPRC in our test environment.

For messages up to 64KB, zHyperWrite is achieving double the rate of the PPRC measurement.

The benefits of zHyperWrite are such that with messages of 128KB, we saw an additional 60 MB per second per log written with a corresponding increase in transaction rate, equating to 120 MB per second **total** data written.

Impact to MQ queue manager costs

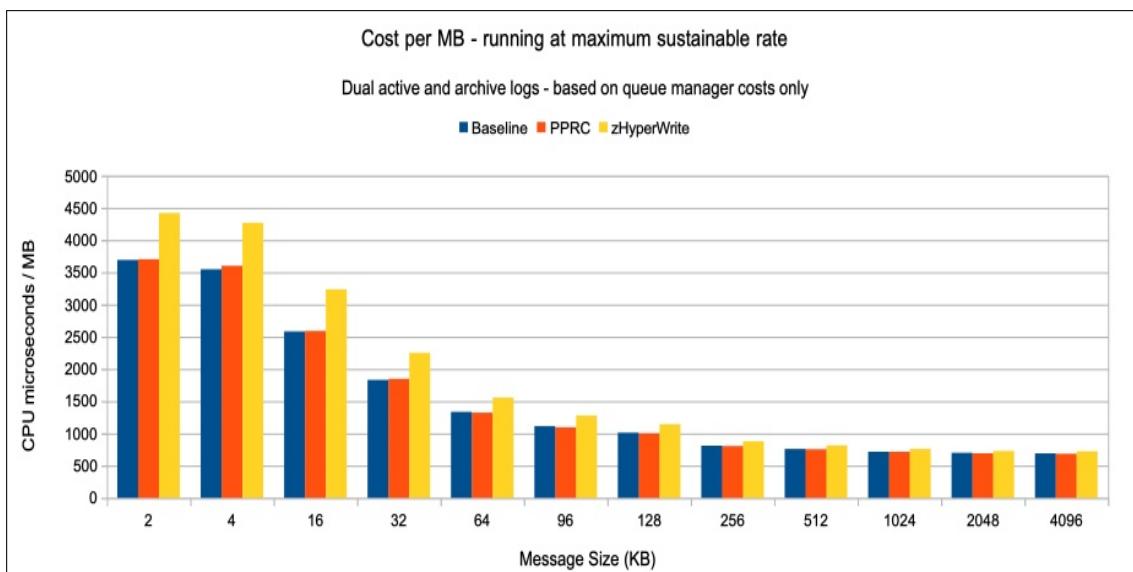
As previously mentioned, the zHyperWrite configuration may see increased SRB costs in the MQ MSTR address space.

There are a number of factors which can affect the actual scale of the impact, and it is difficult to predict exactly how much any particular workload might be affected. Some of the factors include:

- Message size.
- Utilisation of the MQ logger task.
- Amount of data written per I/O.

If we compare the cost per MB in the MQ MSTR address space when running as the maximum sustainable rate, we can plot the following chart.

Chart: Cost to MQ when running at maximum sustained rate



In this example, the maximum increase is using a 16KB workload, zHyperWrite is 25% more expensive than the PPRC configuration.

In these configurations with dual active/archive logs, the cost increase is typically the order of 14%. In a single active/archive log configuration, the increase is more typically 8%.

In the majority of these data points associated with smaller messages, the zHyperWrite configuration is logging at a significantly higher rate, sometimes more than double the rate - which will increase the contention within the queue manager, which in turn can increase costs.

Using a message size where the log rate is similar should minimise any additional costs incurred from higher throughput. In these measurements, the 4MB workload achieves the most similar throughput, but even here there is a 10% increase in log rate in the zHyperWrite configuration.

Table: Comparing costs of 4MB workload

Configuration	Rate MB/Second	Average queue manager CPU (including SRB)	Average queue manager SRB
		CPU uSeconds/MB	CPU uSeconds/MB
Baseline	313	690	383
PPRC	259	684	376
zHyperWrite	283	722	411

This example shows that the zHyperWrite configuration adds 38 CPU microseconds per MB to the PPRC configuration costs. This additional CPU is primarily SRB time and equates to an increase of 5.5% of the total MQ MSTR CPU or 9.3% in the total SRB used by the MQ MSTR address space.

These measurements were made from data gathered when the MQ logger task was fully utilised.

Impact of I/O limitations on dual active and dual archive logs on older hardware

On our current systems using IBM z16 and DS8950F, we did not observe the following issue but we have kept this section as it is likely applicable on systems where non-MQ specific workloads are adding to the load on the DASD subsystems.

When running on MQ queue managers configured with dual active and dual archive logs on previous DS8870 DASD subsystem , the high throughput measurements - typically observed with 256KB messages or larger, saw increased I/O response times due to the saturation of non-volatile storage (NVS) in the I/O subsystem, which manifests itself in the form of increased Disk Fast Write Bypass (DFWBP).

What is Disk Fast Write Bypass?

In the 3990/3390 era, when NVS is full, the write I/O bypasses the NVS and the data is written directly to the disk module (DDM).

In DS8000, when the NVS is full, the write I/O is retried from the host until the NVS space becomes available. So DFW Bypass must be interpreted as DFW Retry for DS8000. If RMF shows that DFW Bypass divided by the total I/O rate is greater than 1%, that is an indication of NVS saturation.

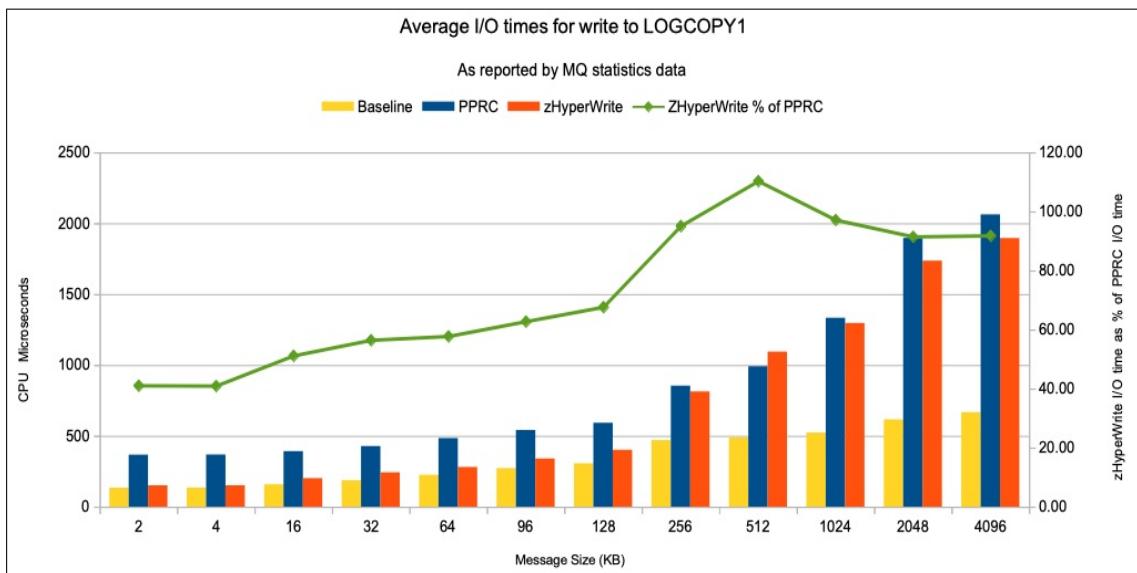
Note that in our measurements, it was not uncommon to see DFW Bypass exceed 80% of the total I/O rate.

Impact on reduced I/O time

The following chart demonstrates the impact on the I/O time from increase DFWBP.

In our configuration, messages of 256KB and larger resulted in the I/O time for the zHyperWrite measurements being much closer to the PPRC measurements due to NVS saturation in excess of 20% - with the 4MB workload seeing DFWBP for in excess of 100% of requests.

Chart: Average I/O time for write to LOGCOPY1



For workloads that are not impacted by DFWBP, i.e. messages up to 128KB, zHyperWrite is taking between 32% and 60% less time per I/O than the equivalent PPRC request.

Impact on reduced I/O time

The chart in this section shows the rate in MB per second that *each* log copy was able to sustain on our system.

In all of these measurements, the queue manager is configured with dual active and dual archive logs, so for every MB put by the application, there are writes of between 4 MB (for baseline) to 6MB (for PPRC or zHyperWrite) in total volume to the disks, i.e.:

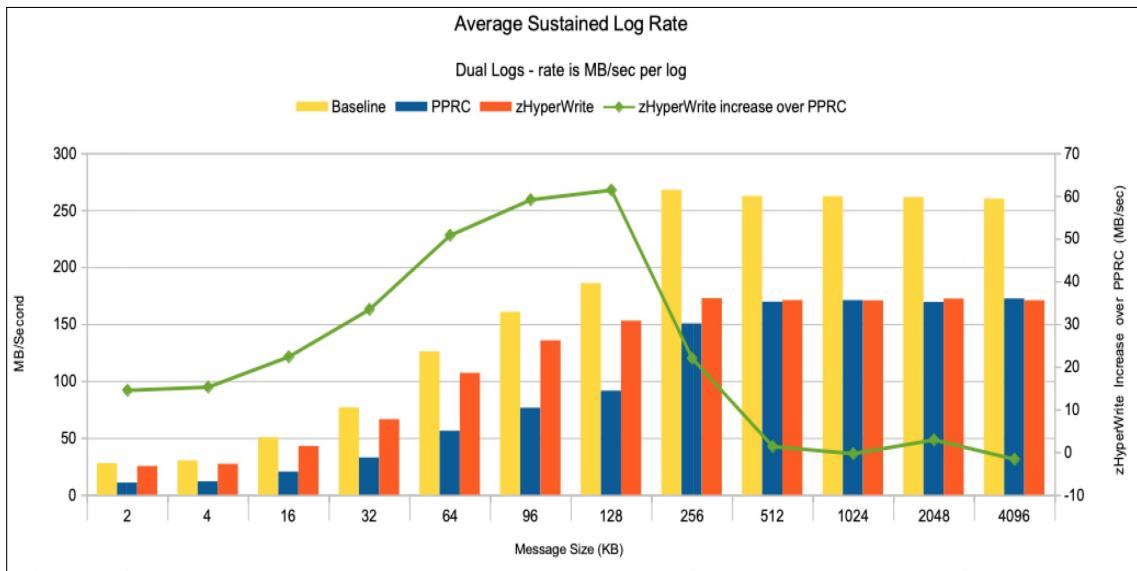
- 2 MB to active log (1 MB per log)
- 2 MB to archives - once the test is running, the queue manager is driving the archive process constantly.
- 2 MB to the mirrored logs (1 MB per log).

What this means is that when the charts are showing 250 MB/second, there is approximately 1GB/second written to the I/O subsystem in the baseline configuration and 1.5GB/second in the PPRC and zHyperWrite configurations.

As discussed previously, at the higher volumes, this causes waits for cache in the I/O subsystem, which is why the performance of PPRC and zHyperWrite becomes similar and the benefits of zHyperWrite are less obvious.

In the case of the PPRC measurement, the system has a separate fibre channel and therefore see less waits for cache - the I/O subsystem is still impacted by DFWBP but not to the same extent as zHyperWrite.

Chart: MQ Log task - sustained log rate with dual logs



Notes on chart:

The green line highlights the difference in MB/second that zHyperWrite offers over PPRC.

For messages of 64KB and less, the zHyperWrite configuration is achieving double the rate of the PPRC measurement.

As noted previously, the benefits of zHyperWrite grow with message size - in this case the actual difference (MB/second) in sustained log rate is peaking with messages sizes between 64KB to 256KB. Workloads with these messages using the zHyperWrite configuration were able to log 50-60 MB/second more than the PPRC equivalent measurement.

After this point, the benefits diminish, largely because the zHyperWrite configuration is being impacted more significantly by I/O cache waits (disk fast write bypass) than the PPRC configuration.

Summary of zHyperWrite benefits

Whilst the zHyperWrite measurements detailed in this report were run on a queue manager configured with single active and single archive logs, this is not the recommended configuration.

Single logs were used to demonstrate the impact of zHyperWrite, particularly when the I/O subsystem can be configured such that any additional I/O load does not cause NVS (Non-Volatile Storage) saturation.

At zero distance replication, zHyperWrite can:

- Reduce the elapsed time from an MQ commit by up to 60%.
- Reduce the elapsed time from an MQPUT by up to 33%.
- Reduce the average I/O time when writing to the MQ active log(s) by up to 60%.
- Improve the maximum sustainable log rate by up to 2.4 times.

Despite the RMF CPU report indicating that the LPAR was less than 25% busy for the entire measurement period, disabling archiving improved the sustained log rate for the 4MB workload by up to 25% for PPRC and 45% for zHyperWrite. This was due in part to periods where the number of CPUs available was insufficient for the number of tasks on the LPAR. Disabling archiving also removed some contention on the channel paths that were used by the writes to the primary volumes of the active logs, as these were also used by the archive process.

As the section “[Impact of I/O limitations on dual active and dual archive logs on older hardware](#)” discusses, the additional load on our previous I/O subsystem when the queue manager was configured with dual active and dual archive logs, coupled with the mirroring of the active logs was enough to result in NVS saturation. This manifested itself in DFWBP (Disk Fast Write Bypass) and resulted in the performance benefits of zHyperWrite being less distinct.

Chapter 5

How It Works

Tuning buffer pools

This chapter gives an outline of how buffer pools are used and what the statistics mean.

Introduction to the buffer manager and data manager

This describes how buffer pools are used. It will help you determine how to tune your buffer pools.

The data manager is responsible for the layout of messages within one or more 4KB pages, and for managing which pages are used on a page set. A message always starts on a new page and a long message can span many pages. A page can contain persistent messages or non-persistent messages, but not both.

The buffer manager is responsible for reading and writing these pages to the page sets, and for managing copies of these pages in memory. The buffer manager makes no distinction between persistent and non-persistent messages, so both persistent and non persistent messages can be written to the page set.

A buffer pool page is written to a page set at the following times:

- At checkpoint, if it contains any change since it was last written and this is the second checkpoint to have occurred since the first such change.
- Whenever the threshold of less than 15% free buffer pool pages is reached. Pages are then written asynchronously by an internal task. This is referred to as the "15% free threshold".
- When an application has finished with a page and there are less than 5% free pages in the buffer pool.
- At shutdown, if it contains any change since it was last written.
- From V5.3 buffer pool pages which contain non-persistent messages are usually not written to a page set at checkpoint or shutdown.

A page is changed both when a message is put and when it is retrieved, because the MQGET logically deletes the message unless it is a browse.

Pages are usually written by the Deferred Write Process (DWP, although it is sometimes called DWT) asynchronously from user application activity. The DWP writes pages from the buffer pool in least recently used order (that is, from the oldest changed page).

A page is read from a page set data set into the buffer pool at the following times:

- When a message that is not already in the buffer pool is required.
- During read ahead, which is when an internal task reads a few messages into the buffer pool before an application needs them. This happens if the current MQGET does I/O to read a page and was not using MSGID or CORRELID.

Read ahead is most effective when you have a few applications getting short persistent messages with only a few messages per unit of work, because the read ahead is more likely to complete while the application waits for log I/O.

There is no direct user control on read ahead. However, you might be able to improve throughput and response time by using multiple queues on separate page set data sets spread across multiple volumes to reduce I/O contention.

Differences in performance due to the size of a buffer pool depend on the amount of I/O activity between a buffer pool and the associated page set data sets. (Real storage usage, and hence paging, might also be a factor but this is too dependent on individual system size and usage to be usefully discussed here.) The unit of I/O is a page rather than a message.

The effect of message lifespan

This section discusses some message usage scenarios.

- For messages that are used soon after they are created (that is, typically within a minute, but possibly up to 2 checkpoint intervals) and a buffer pool that is large enough to contain the high water mark number of messages, plus 15% free space:
 - Buffer pool pages containing such messages are likely to be re-used many times, meaning that relatively few pages need to be written at checkpoint and almost no pages need to be read.
 - Both CPU cost and elapsed time are minimized.
- For messages that are stored for later batch processing:
 - All pages containing such messages are likely to be written to the page set data set because they become more than 2 checkpoints old, regardless of buffer pool size. All these pages need to be written again after the messages are used by an MQGET call, for example at the second checkpoint after the MQGET call (because the pages on the page set still contain the messages and must eventually reflect the fact that the messages are now flagged as deleted). However, if pages are reused for new messages before being written to the page set, one write operation will cover the MQGET of the old messages and the MQPUT of the new.
 - MQGET operations can still be satisfied directly from the buffer pool, provided that the pool has not reached the 15% free threshold since the required message was MQPUT.
- In either case, if the 15% free threshold is crossed, the DWP is started. This uses the least recently used algorithm for buffer pool pages to write the oldest changed buffer pool pages to the page set and make the buffer pool pages available for other messages. This means that any messages written to a page set will have to be read back from the page set if the buffer pool page is reused.
 - This is the least efficient buffer pool usage state. Elapsed time and CPU cost will be increased.
 - In many cases (for example, a single queue that is much larger than the buffer pool and is accessed in first in first out sequence) most messages will have to be read from the page set.

- A busy buffer pool, once in this state, is likely to remain so.
Non-persistent message processing does not require IBM MQ log I/O and thus page set read I/Os might have greater impact on elapsed time.

A buffer pool that is large enough to keep 15% free buffers will avoid any reads from the page set (except after a queue manager restart).

Understanding buffer pool statistics

A page in a buffer pool is in one of five states

Unused

This is the initial state of all pages within the buffer pool.

Changed and in use

The content of the page in the buffer pool is different from the matching page on the page set. Eventually the queue manager will write the pages back to the page set. The page is currently in use by an application, for example a message is being placed within it. When a large message is being put, many pages might be updated, but usually only one page will be in use at a time.

Changed and not in use

The page is the same as "Changed and in use" except that the page is not in use by an application.

Unchanged and in use

The content of the page in the buffer pool is the same as the matching page on the page set. The page is in use, for example, an application is browsing a message on the page.

Unchanged and not in use

The content of the page in the buffer pool is the same as the matching page on the page set, and the page is not in use by an application. If a buffer for a different page is required, the buffer page can be reassigned without its contents being written to disk.

- The term stealable buffers refers to those buffers that are unused or unchanged and not in use. The number of stealable buffers available as a percentage of the total number of buffers affects the behavior of the buffer pool.
- A page can only be written to disk if it is 'changed' and not 'in use'. In some circumstances, pages that are 'changed' and 'in use' are written to disk synchronously after the application has finished with the page - which results in the page becoming 'changed' and not 'in use' only when the I/O completes.
- When a changed page is written to disk (so the version on disk is the same as that in the buffer pool) the page becomes unchanged and not in use.

The data manager issues requests for pages to the buffer manager. If the contents of a page are required, a request to get a page is issued:

- The buffer manager checks to see if the page is already in the bufferpool; if it is, the page status is set to in use and the address of the page is returned.
- If the page is not in the buffer pool, a stealable buffer is located, the page status is set to in use, the page is read in from the page set, and the address of the page is returned.
- If an update is going to be made to the page, the data manager calls the buffer manager with a SET WRITE request. The page is then flagged as changed and in use.
- When an application has finished with the page it releases it and, if no other application is using the page, the status is changed to not in use.

If the contents of the page are not required (for example, the page is about to be overwritten with a new message) a request to get a new page is issued. The processing is the same as above except, if the requested page is not in the buffer pool, a stealable buffer is located but the page is not read in from the page set.

Definition of buffer pool statistics

This section describes the buffer pool statistics. The names given are as described in the assembler macro thlqual.SCSQMACS(CSQDQPST) and is discussed in more detail in the InfoCenter section 'Buffer manager data records'. The names shown in brackets are those used by the program MQSMF which can print out SMF statistics. (MQSMF is available as part of performance report [MP1B](#) "Interpreting accounting and statistics data").

QPSTNBUF(#buff)	The number of pages allocated to the buffer pool in the CSQINP1 data set at MQSeries startup.
QPSTCBSL(#low)	The lowest number of stealable buffers during the SMF interval.
QPSTCBS(#now)	The number of stealable buffers at the time the SMF record was created.
QPSTGETP(getp)	The number of requests to get a page that were issued.
QPSTGETN(getn)	The number of requests to get a new page that were issued.
QPSTSTW(STW)	The number of SET WRITE requests that were issued.
QPSTRIO(RIO)	The number of pages that were read from the page set.

If the percentage of stealable buffers falls below 15% or the percentage of changed buffers is greater than 85%, the DWP is started. This task takes changed pages and writes them to the page sets, thus making the pages stealable. The task stops when there are at least 25% stealable pages available in the buffer pool.

When the status of a changed page goes from in use to not in use, and the percentage of stealable pages falls below 5% or changed pages is greater than 95%, the page is written to the page set synchronously. It becomes unchanged and not in use, and so the number of stealable buffers is increased.

When a checkpoint occurs, all pages that were first changed at least two checkpoints ago are written to disk, and then flagged as stealable. These pages are written to reduce restart time in the event of the queue manager terminating unexpectedly.

If a changed page was in use during checkpoint processing or when the DWT ran, but should have been written out, the page is written out to disk synchronously when the page changes from in use to not in use.

QPSTDWT(DWT)	The number of times the DWP was started.
QPSTTPW(TPW)	The total number of pages written to page sets.
QPSTWIO(WIO)	The number of write request.
QPSTIMW(IMW)	The number of synchronous write requests. (There is some internal processing that periodically causes a few pages to be written out synchronously.)
QPSTDWC(DMC)	The number of times pages were written synchronously to disk because the percentage of stealable buffers was less than 5% or changed pages was greater than 95%.

When the data manager requests a page that is not in the buffer pool, a stealable page has to be used.

QPSTSTL(STL)	The number of times a page was not found in the buffer pool and a stealable page was used.
QPSTSOS(SOS)	The number of times that a stealable page was needed and there were no stealable pages available (a short on storage condition).
QPSTSTLA(STLA)	The number of times there was contention when getting a stealable page.

Interpretation of MQ statistics

1. If QPSTSOS, QPSTSTLA, or QPSTDPMC are greater than zero you should increase the size of the buffer pool or reallocate the page sets to different buffer pools.
2. For buffer pool 0 and buffer pools that contain short lived messages:
 - QPSTDWT should be zero and so the percentage QPSTCBL/QPSTNBUF should be greater than 15%.
 - QPSTTPW might be greater than 0 due to checkpointing activity.
 - QPSTRIO should be 0, unless messages are being read from a page set after the queue manager is restarted.
 - A value of QPSTSTL greater than 0 indicates that pages are being used that haven't been used before. This could be caused by an increased message rate, messages not being processed as fast as they were (so there is a build up of messages), or larger messages being used.
 - You should plan to have enough buffers to handle your peak message rate.
3. For buffer pools with long lived messages, where there are more messages than will fit into the buffer pool:
 - (QPSTRIO+QPSTWIO) Statistics interval is the I/O rate to page sets. If this value is high, you should consider using multiple page sets on different volumes to allow I/O to be done in parallel.
 - Over the period of time that the messages are processed (for example, if messages are written to a queue during the day and processed overnight) the number of read I/Os (QPSTRIO) should be approximately the total number of pages written (QPSTTPW). This shows that there is one disk read for every page written.
If the QPSTRIO is much larger than QPSTTPW, this shows that pages are being read in multiple times. This could be caused by application using MQGET by MSGID or CORRELID, browsing messages on the queue using get next, or using message selectors. The following actions might relieve this problem
 - Increase the size of the buffer pool so that there are enough pages to hold the queue, in addition to any changed pages.
 - Move page sets to a different buffer pool to reduce contention between messages from different applications.
 - Use the INDXTYPE queue attribute which allows a queue to be indexed by MSGID or CORRELID and eliminates the need for a sequential scan of the queue.
 - Change the design of the application to eliminate the use of MQGET with MSGID or CORRELID, or the get next with browse option. Applications using long lived messages typically process the first available message and do not use MQGET with MSGID or CORRELID, and they might browse only the first available message.

Example of a badly tuned buffer pool

This example was taken from a production system. Buffer pool 0 contains only page set 0.

The system was being monitored using the ISPF interface on TSO to display information about queues and channels. The initial symptom was that throughput to the distributed MQ systems dropped by a factor of 100.

Table: Buffer pool statistics for intervals		
Field	Previous interval	Problem interval
QPSTNBUF	1050	1050
QPSTCBSL	300	154
QPSTCBS	308	225
QPSTGETP	1800000	23000000
QPSTGETN	16000	13000
QPSTRIO	0	310000
QPSTSTW	508000	432000
QPSTTPW	940	1938
QPSTWIO	59	107
QPSTIMW	29	47
QPSTDWT	0	11
QPSTDPMC	0	0
QPSTSTL	84	732000
QPSTSTLA	0	421000
QPSTSOS	0	0

Observations on the problem interval

1. The value for QPSTSTLA (contention when getting a stealable buffer) is 421000. This is extremely high.
2. More than half the request for a stealable buffer had contention
 $(\text{QPSTSTLA}/\text{QPSTSTL}) = 421000/732000$.
3. The number of pages read (QPSTRIO) is very high. 310,000 I/O in 30 minutes is approximately 172 I/O per second (about the maximum capacity of the device).
4. QPSTDPMC is zero so the buffer pool was not critically short of buffers.
5. QPSTDWT is greater than zero, QPSTCBSL/QPSTNBUF=154/1050 is 14.6%, QPSTTPW=1938, these figures are not unusual.
6. QPSTGETN is lower than the previous interval, but QPSTGETP is significantly higher. Also QPSTSTW is lower, indicating less updates. This implies that there were more requests for MQGET with browse or by MSGID or CORRELID.

What was happening

1. In the mover, information on channels is held in messages on the SYSTEM.CHANNEL.SYNC.QUEUE. At the end of a batch, the messages relating to the channel are retrieved from the queue. The MQGET request uses MSGID which is the index type on the queue in the sample.
2. The SYSTEM.CHANNEL.SYNC.QUEUE was in page set 0 and in buffer pool 0.
3. Normally there were sufficient stealable pages for the whole of the SYSTEM.CHANNEL.SYNC.QUEUE to be kept in the buffer pool.
4. The model queue definitions for command requests and responses pointed to page set 0.
5. For some reason (perhaps the ISPF operator asked for all information about all queues, which produced many response messages) buffer pool 0 filled up.
6. DWT processing moved the older pages out to disk and made the pages stealable.
7. When a channel reached the end of a batch, it had to read pages for the channel from the page set looking for a particular message. Because there were insufficient stealable buffers to hold the whole of the SYSTEM.CHANNEL.SYNC.QUEUE in the buffer pool, stealable pages were reused and so, for example, the buffer that held the first page of the queue was reused and was replaced with the 100th page of the queue.
8. When the next channel reached the end of a batch, it had to read the first page of SYSTEM.CHANNEL.SYNC.QUEUE from disk and re-use a stealable buffer. The stealable buffers were then "thrashing".
9. In time, the problem would gradually have corrected itself as pages on the SYSTEM.CHANNEL.SYNC.QUEUE became changed when messages were put to and retrieved from the queue. However the ISPF panels were used to display information about the system, and pages were being written out to disk again, and the whole cycle repeated itself.

Actions taken to fix the problem

1. The SYSTEM.COMMAND.REPLY.MODEL queue was altered to use a storage class on a different page set, and so in a different buffer pool.
2. The size of buffer pool 0 was doubled. This was not strictly necessary but it allowed room for any unexpected expansion.

Log manager

The log manager is responsible for writing recovery information to the log data sets. This information is used to recover in the event of a failure or a request to roll back recoverable changes. Recoverable resources includes persistent messages and MQ objects. Non-persistent messages are not recoverable and are not handled by the log manager; they are lost at system restart.

This section discusses only recoverable resources.

The log is conceptually a very long buffer. In practice the log is implemented using virtual storage and DASD. The position of information in this buffer is defined by the Relative Byte Address (RBA) from the start of the buffer.

Description of log manager concepts and terms

This section describes the concepts and terms used in this section. They are described more fully in the IBM MQ for z/OS Concepts section of the IBM Knowledge Center.

- Each log buffer is 4096 bytes long and resides in virtual storage.
- The number of log buffers is determined from the OUTBUFF keyword of the CSQ6LOGP macro.
- When the log buffers fill, or an application issues a commit, the buffers are moved from virtual storage to log data sets, called the active log data sets. When the log records have been written, the log buffers can be reused.
- There are at least two active log data sets, which are used cyclically.
- Dual logging describes the situation where the log buffers are written to two log data sets. In the event of the loss of one data set, the other data set can be used. This facility is enabled with the TWOACTV keyword of the CSQ6LOGP macro.
- Single logging is when only one ring of active data sets are used.
- When an active log data set fills up, an archive log data set is allocated and the active log is copied to it. When the copying has completed, the active log data set can then be reused.
- A data set called the bootstrap data set (BSDS) records which RBA range is on which active or archive log. At system restart, the BSDS is used to identify which log data set to use first.
- You can have two copies of the BSDS data set, so in the event of the loss of one BSDS, the other can be used.
- When an active log is archived, the BSDS data sets are also archived.

Other terms used in this description

- The current log buffer is the log buffer that is being filled. When this buffer fills up, the next buffer is used and becomes the current log buffer.
- The logger is a task, running within the queue manager, that handles the I/O to log data sets.
- A log check request occurs while work is being committed. If the data up to the RBA has not been written out to disk, a request is made to the logger passing the RBA value, and the requester is suspended. The logger writes the data up to the RBA out to disk and resumes any tasks waiting for the logger. When the log check request completes, the data has been copied to disk and it can be used in the event of a failure. A log check is issued when:
 - A commit is issued.
 - A persistent message is put or got out of syncpoint.

- o An MQ object, such as a local queue, is defined, deleted or altered.

Illustration of logging

The following section gives a simplified view of the data that is logged when an application gets a persistent message and issues a commit.

When a message is got, a flag is set to indicate that the message is no longer available. The change to the flag and information to identify the message within the page, along with information to identify the unit of work, are written to the log buffers. During the commit, "end of unit of work" information is written to disk and a log check request is issued with an RBA of the highest value used by the application.

When does a write to the log data set occur?

Log buffers are written to disk at the following times:

- When a log check request is issued. When the application is running under a syncpoint coordinator (for example, CICS Transaction Server) and has issued update requests to multiple resource managers (such as MQ requests) and recoverable CICS resources, the sync level 2 protocol is used at commit time. This causes two MQ log check requests, one for the PREPARE, and the other for the COMMIT verbs.
- If the number of filled log buffers is greater than or equal to the value of WRTHRSH specified in the CSQ6LOGP macro, a request is made to the logger to write out up to the RBA of the previous page.
- When all of the log buffers are in use and there are none free.
- When the system shuts down.

The logger writes up to 128 log buffers at a time to the log data sets, so 129 log buffers require at least two I/O requests, but the buffers might be written out when other applications are issuing log check requests.

How data is written to the active log data sets

The current log buffer is the buffer that is currently being filled with data. When this buffer fills up, the next buffer is used and becomes the current log buffer.

Single logging

If the log check request specifies an RBA that is not in the current buffer, the logger writes up to and including the page containing the specified RBA.

If the log check request specifies an RBA that is in the current buffer, the logger writes any log buffers up to, but not including, the current buffer, and then writes the current buffer up to the requested RBA (a partial page) with another I/O.

Dual logging

If the log check request specifies an RBA that is not in the current log buffer, the I/Os are performed on each log data set in parallel.

If the check request specifies an RBA in the current buffer, the I/Os will be performed on each log data set in parallel unless the DASD used does not support write caching. Should DASD without write caching be used, any rewrites of the buffer writes will be performed to each log data set in series.

Rule of thumb In effect, for a log check request with dual logging, the elapsed time for the write of the current page to the log data sets is the time required for two I/Os in series; all other log writes take the time for one log I/O.

Interpretation of key log manager statistics

Consider an application that gets two messages and issues a commit.

When a message is retrieved, a flag is set to indicate that the message is no longer available. The change to the flag and information to identify the message within the page, along with information to identify the unit of work, are written to the log buffers. The same happens for the second message. During the commit, "end of unit of work" information is written to disk and a log check request is issued with an RBA of the highest value used by the application. data records'. If this is the only work running in the queue manager, the three log requests are likely to fit in one log buffer. The log manager statistics (described in the IBM Knowledge Center section 'Log manager system data records) would show the following:

QJSTWRNW	Number of log writes with no wait	3
QJSTBFFL	Number of log pages used	1
QJSTBFWR	Number of calls to logging task	1
QJSTLOGW	Number of I/O to each log data set	1

In reality, more data than just one flag is logged and there might be more than one I/O involved. This is explained below.

Detailed example of when data is written to log data sets

Consider two applications, each putting a persistent message and issuing a commit. Assume that:

- Each message needs 16 log buffers to hold all of the data
- The WRTHRSH value in CSQ6LOGP is 20
- Dual logging is used

The following figure shows the log buffers used:

Message 1	Message 2	Commit 1	Commit 2
B1 B2 ... B15 B16	B17 B18 ... B31 B32	B33	B34

Where:

- B1...B16 are the 16 log buffers for message 1
- B17...B32 are the 16 log buffers for message 2
- B33 is the log buffer for the commit of the first application
- B34 is the log buffer for the commit of the second application. In reality, each log buffer usually contains information from different applications, so an individual log buffer might contain information from message 1 and message 2.

If the interval between each MQPUT and the commit is relatively long compared to the time taken to do a disk I/O (for example, more than 20 milliseconds), the following happens:

1. The first message is put, buffers B1-B16 are filled.
2. When the second message is being put, and buffer B21 is about to be filled, because the number of full log buffers is greater than the value of WRTHRSH in CSQ6LOGP, this signals the logger to write out pages up to (but not including) the current buffer. This means that

buffers B1-B20 are written out, buffers B1-15 in one I/O, and buffers B16-B20 in a second I/O.

3. When buffer B22 is being filled, the number of full log buffers is greater than WRTHRSH so a request is made to the logger, passing the RBA of page B21. Similarly, when writing B23 a request is made to the logger to write out buffer B22.
4. When the I/O to buffers B1-B15 has completed, these buffers are available for reuse, and so the number of full buffers falls below the value in WRTHRSH and no more requests are made to the logger.
5. When buffer B23 is being filled, the number of full log buffers is not greater than WRTHRSH, so a request is not made to the logger.
6. When the logger has finished processing the requests for buffers B1-15 and B16-20, it checks the work on its input queue. It takes the highest RBA found and writes up to that page to the data sets (so it would write out pages B21-B22). In practice, all of the buffers B23-B32 would be filled while the I/O of buffers B1-B15 is happening.
7. When commit 1 is issued, a log check is issued and buffers B23-B32 are written out in one I/O and buffer B33 (the current buffer) written out in a second I/O. The I/O for buffers B21-B32 is performed in parallel, and because this is the first time B33 has been written, the I/O is performed in parallel. The time taken for the commit is at least the time to perform two I/Os.
8. When commit 2 is issued, buffer B33 is rewritten, so the I/O is performed in series. Buffer B34 (the current buffer) is written out and the I/O to the two logs is performed in parallel. This commit request takes at least the time to do three I/O requests. When B34 is rewritten, the I/O is performed in series.

If the interval between the MQPUTs and the commits is very short compared to a disk I/O (for example less than 5 milliseconds), the following happens:

1. As before, when the second message is being put, and buffer B21 is about to be filled, because the number of full log buffers is greater than the value of WRTHRSH in CSQ6LOGP this signals the logger to write out pages up to (but not including) the current buffer. Buffers B1-B20 are written out, buffers B1-15 in one I/O, and buffers B16-B20 in a second I/O. The I/Os to each log data set are done in parallel.
2. If both the commits are issued while the above I/Os are happening, when the I/Os have finished, the logger writes buffers B21-B33 out in one I/O and buffer B34 (the current buffer) in a second I/O. The I/O for buffers B21-B33 is done in parallel, and the I/O for the current log buffer (B34) is also done in parallel to the two log data sets. The next time buffer B34 is rewritten, the I/O is done in series. The following table summarizes which buffers are written in each I/O:

Long interval	Short interval
B1...B15 in parallel	B1...B15 in parallel
B16...B20 in parallel	B16...B20 in parallel
B21...B22 in parallel	B21...B33 in parallel
B23...B30 in parallel	B34 in parallel
B33 in parallel	
B33 in series	
B34 in parallel	
Time taken: 8 I/O.	Time taken: 4 I/O. However, because more data is written in each I/O on average, each I/O takes longer than the long interval case.
The next log check request rewrites B34 in series	The next log check request fewrites B34 in series.

The effect of one log I/O containing data from multiple transactions is called coat-tailing. As a general rule, as the amount of data written to the log increases, the response time of requests requiring a log check increases.

In the example above, if the value of OUTBUFF was 80 (giving 20 log buffers) the put of message 2 would be suspended just before the write of buffer 21 because there are no free log buffers because buffers B1-B20 are all in use, with buffers B1-B15 being written to the log data sets. When the I/O completes and buffers B1-B15 are available again, the application can be resumed. The number of times that an application is suspended because no buffers are available is recorded in the log manager statistic QJSTWTB. If you get a nonzero value in this field, you should increase the value of OUTBUFF until the value of QJSTWTB remains at zero.

MQPUT example

Table: Interpretation of log statistics from MQPUT and commit of 100,000-byte messages		
QJSTWRNW	Number of log writes with no wait	215
QJSTBFFL	Number of log pages used	2 550
QJSTBFWR	Number of calls to logging task	200

- The information in the table is for 100 messages, so each message used approximately 25 log pages per message. Each log page is 4096 bytes long, so the 25 pages use 102,400 bytes. This includes the information about which pages have been changed, and information about the unit of work.
- For each MQPUT and commit there were two calls to the logging task, one call was made because the number of full log buffers was greater than the value of WRTHRSH (20), the other call was made during the commit.
- To write out 25 pages causes one I/O for 15 pages, another I/O for 9 pages, and an I/O for the current log buffer. The elapsed time taken to log the data is the duration of 4 I/Os, the parallel I/O for the 15 pages and the 9 pages, and two I/Os in series for the current log buffer.

MQGET example

Table: Interpretation of the log statistics from MQGET and commit of 100 000-byte messages		
QJSTWRNW	Number of log writes with no wait	110
QJSTBFFL	Number of log pages used	29
QJSTBFWR	Number of calls to logging task	102

- The information in the table is for 100 messages so there is approximately one call to the logger per message.
- Only 29 pages were used to hold the log data. This shows that not very much data was logged and the same page used for several requests before the page was full.
- The same page was written out several times, even though it had not been completely filled.
- Because the current log buffer only was written each time, there was one I/O to each log, and because it was for the current buffer, these I/O were done in series.

Interpretation of total time for requests

In some measurements, the time taken to put a 100 000-byte message and a commit was 67 milliseconds on average, and the time to get a 100 000-byte message and a commit was 8 milliseconds on average. In both cases, most of the elapsed time was spent waiting for log I/O.

For the MQGET, the write I/Os to the dual logging devices were done in series. Because little data was written each time the connect time, when data was transmitted to the device, was small and RMF reports showed that the device had a short response time of 3-4 milliseconds. Two I/Os taking 3-4 milliseconds is close to the time of 8 milliseconds.

For the MQPUT, the write of the 15 and the 9 pages were done in parallel, and the write of the current buffer were done in series; in effect the time taken for four I/Os. Because a lot of data was written in a request, this caused a longer connect time, which leads to a longer overall DASD response time. RMF showed a response time of about 16-17 milliseconds for the log DASD. Four I/Os of 16-17 milliseconds is close to the measured time of 67 milliseconds.

What is the maximum message rate for 100 000-byte messages?

If we assume that:

- Put and commit of 100 messages use 2550 buffers (from the figures above)
- Get and commit of one message uses less than 1 buffer
- MQ writes a maximum 128 buffers for every I/O
- The I/O response time when writing 15 buffers per I/O was about 20 milliseconds
- The I/O response time for writing the current log buffer was 4 milliseconds
- There were no delays when writing to DASD (this includes within zOS and within the DASD subsystem)
- Concurrent copies of an application which puts a message, commits, gets the message, and commits again. We can estimate the maximum message rate as follows:
 1. Out of the 2550 log buffers used for MQPUTs, 100 are written as the current log buffer, so 2450 can be written in parallel
 2. We can write up to 15 pages per I/O, so 2450 pages need 164 I/Os
 3. 164 I/Os, each taking 20 milliseconds gives a total of 3280 milliseconds

4. Each commit writes the current log buffer to each log data set in series. There are 100 commits for puts and 100 commits for gets. For two I/Os in series, each of 4 milliseconds, the total time for writing the current log buffers is $(100 + 100) * 2 * 4$ giving a total of 1600 milliseconds.
5. Total time for the I/O is $3280 + 1600$ giving a total of 4880 milliseconds.
6. If it takes 4.88 seconds to process 100 messages, 20.5 messages could be processed in 1 second. This means that the theoretical absolute message rate is 20.5 messages per second.

This is the theoretical maximum with the given assumptions. In practice, the maximum will be different because the assumptions made are not entirely realistic. In a measurement made using a requester/reply application model where a CICS transaction put a 100 000-byte message to a server, and received the same message back, the transaction rate was 10-11 transactions (21 messages) per second.

Chapter 6

Advice

Use of LLA to minimize program load caused throughput effects

IBM MQ sometimes needs to load programs when applications or channels start. If this happens very frequently then the I/O to the relevant program libraries can be a significant bottleneck.

Using the Library Lookaside (LLA) facility of the operating system can result in very significant improvement in throughput where program load I/O is the bottleneck.

The member CSVLLAxx in SYS1.PARMLIB specifies the LLA setup. The inclusion of a library name in the LIBRARIES statement means that a program copy will always be taken from VLF(Virtual Lookaside Facility) and hence will not usually require I/O when heavily used. Inclusion in the FREEZE statement means that there is no I/O to get the relevant DD statement concatenation directories (this can often be more I/O than the program load itself). Use the operating system “[F LLA,REFRESH](#)” command after any changes to any of these libraries.

The following are some specific examples of when programs are loaded:

Frequent use of MQCONN/MQDISC - for example WLM Stored Procedures

Every time an MQCONN is used, an IBM MQ program module has to be loaded. If this is done frequently then there is a very heavy load on the STEPLIB library concatenation. In this case it is appropriate to place the SCSQAUTH library in the CSVLLAxx parmlib member LIBRARIES statement and the entire STEPLIB concatenation in the FREEZE statement.

For example: Ten parallel batch applications running on the same queue manager were used to drive WLM (Work Load Manager) stored procedures, where each application looped 1000 times issuing ‘EXEC SQL CALL Stored_Proc()’. All stored procedures ran in a single WLMPAS address space. The stored procedures issued MQCONN, MQOPEN, MQPUT (a single 1K nonpersistent message), MQCLOSE, MQDISC, but no DB2 calls were made, and were linked with the MQ/RRS stub CSQBRSTB.

1. We achieved 300 transactions a second with all of the WLMPAS’s STEPLIB concatenation in LLA (in both the LIBRARIES(..) and FREEZE(..) dataset lists of the parmlib member CSVLLAxx)
2. We achieved 65 transactions a second with just the LIBRARIES(..)
3. We achieved 17 transactions a second without any such tuning

Frequent loading of message conversion tables

Each conversion from one code page to another requires the loading of the relevant code page conversion table. This is done only once per MQCONN, however, if you have many batch programs instances which process only a few messages each then this loading cost and elapsed time can be minimised by including the STEPLIB concatenation in both the LIBRARIES(..) and FREEZE(..) lists.

Frequent loading of exits - for example, channel start or restart after failure

Channels can have a number of separate exits; SCYEXIT MSGEXIT, SENDEXIT and MREXIT for MCA channels, and SCYEXIT, SENDEXIT, RCVEXIT for SVRCONN channels. If a significant number of channels start in a short time then a heavy I/O requirement is generated to the exit libraries.

In this case the CSQXLIB concatenation must be included in the FREEZE(..) dataset lists to gain any benefit as a BLDL is done for every exit for every channel.

Frequent loading of CSQQDEFV

Each time IMS checks to see if it can connect to a particular queue manager, it has to load the CSQQDEFV module. By placing the dataset in both the LIBRARIES(..) and FREEZE(..) lists we saw an improvement in transaction rate of up to 10%.

System resources which can significantly affect IBM MQ performance

Clearly, having more than enough CPU power is desirable in any system.

DASD I/O capability, particularly write response time and data rate through DASD controller NVS can significantly affect the IBM MQ log and hence persistent message throughput. DASD I/O capability, particularly page set reads for MQGET, can affect performance where large amounts of message data require messages to be spilled from buffer pools and later read back from page sets.

For shared queues,

- CF link type can significantly affect performance. ICP links are faster than CBP links which are faster than CFP links.
- Enough CF CPU power must be allowed for, remembering that it is recommended not to exceed 60% busy on a CF.
- CFLEVEL(4) application structures can use DB2 table and log resources for which DASD I/O capability is important for performance.

Large Units of Work

Multiple messages can be processed in a single unit of work, i.e. within the scope of a single commit. Unless changed at your site, the default setting of MAXUMSGS is 10,000 and can be reviewed using the “DISPLAY QMGR MAXUMSGS” command.

Using larger units of work can use additional storage. In the case of messages on private queues, additional storage will be used from the queue managers private storage, whereas shared queue messages will use storage in the CSQ_ADMIN structure.

The MQPUT cost is typically consistent for units of work up to 10,000 messages.

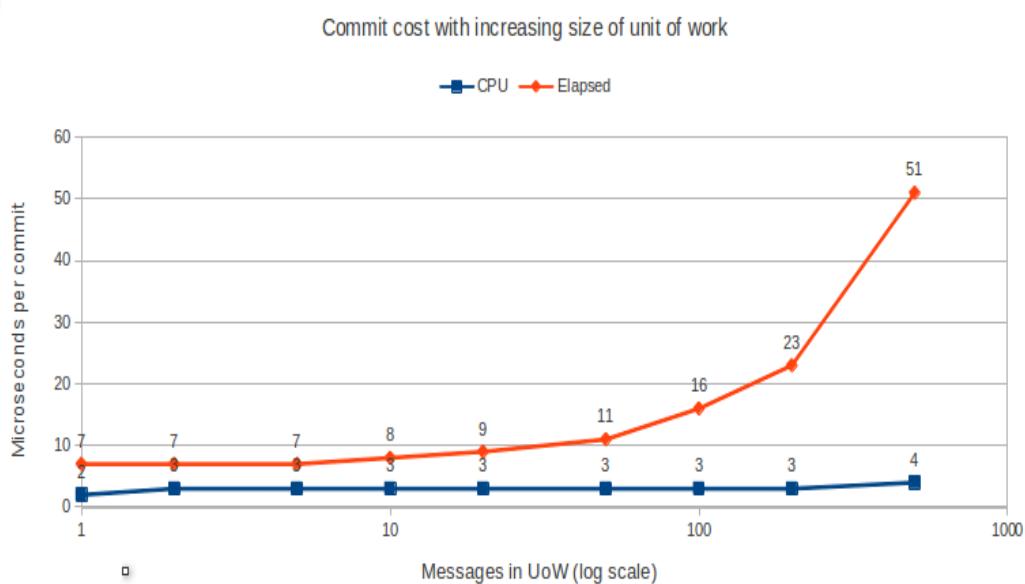
The MQGET cost is typically consistent for units of work up to 10,000 messages except:

- When the queue is indexed and the messages use a common value for the index. In this case, whether using get-next or get-specific there can be additional cost incurred when the unit of work is exceeds 1000 messages.
- If the gets are running at the same time as puts, there will be interaction. For example a task tries to get a message from a queue whilst another task is putting a large number of messages in a unit of work. As the messages being put are in a unit of work, they are not available. This results in the getter task attempting to get each message on the queue. If there are 100,000 messages in the unit of work, the getter task may attempt up to 100,000 gets. If each of these gets takes 1 microsecond, the failed get could cost 100 milliseconds of CPU.

Typically the time of taken to commit will increase as more data needs to be committed. As the number of messages per commit increases, the duration of the commit time is spread across more messages, so the impact is reduced as the number of messages per commit increases.

The following chart shows the CPU and elapsed time per commit as the number of messages in a unit of work increases.

Note: When there is 1 message per commit, for 1000 messages there will be 1000 commits. This means that the commit cost would be 2000 microseconds and the elapsed time would be 7000 microseconds. Contrast this 1000 messages per commit where the commit cost is 4 microseconds and the elapsed time is 51 microseconds.



Application level performance considerations

If and when appropriate consider:

- Is there really a requirement to use persistent messages?
- Processing multiple reasonably small messages in one unit of work (that is, within the scope of one commit)
 - But do not usually exceed 200 messages per unit of work
- Combining many small messages into one larger message, particularly where such messages will need to be transmitted over channels. In this case a message size of 30KB would be sensible.
- The following are particularly important for thin client applications
 - Minimise the number of calls, in particular keep connected, and keep a queue open.
 - Is it necessary to use temporary dynamic queues?
 - Use MQPUT1 rather than MQOPEN, MQPUT, MQCLOSE, unless multiple MQPUTs are going to be issued against the same queue.
- Application requirement for strict message ordering.
- If using IMS message processing regions, preload the IBM MQ modules.

Some applications require messages to be processed in strict order of arrival. This requires that a **single** application instance process a queue of messages. There would be many requester applications but only a single reply application. If this application should fail or slow down due to resource contention, the queue depth could rise dramatically.

Common MSGID or CORRELID with deep shared queues

MQ shared queues can contain many millions of messages and with the use of unique index keys, such as message ID, can enable the application to get specific messages from the queue with consistent response times.

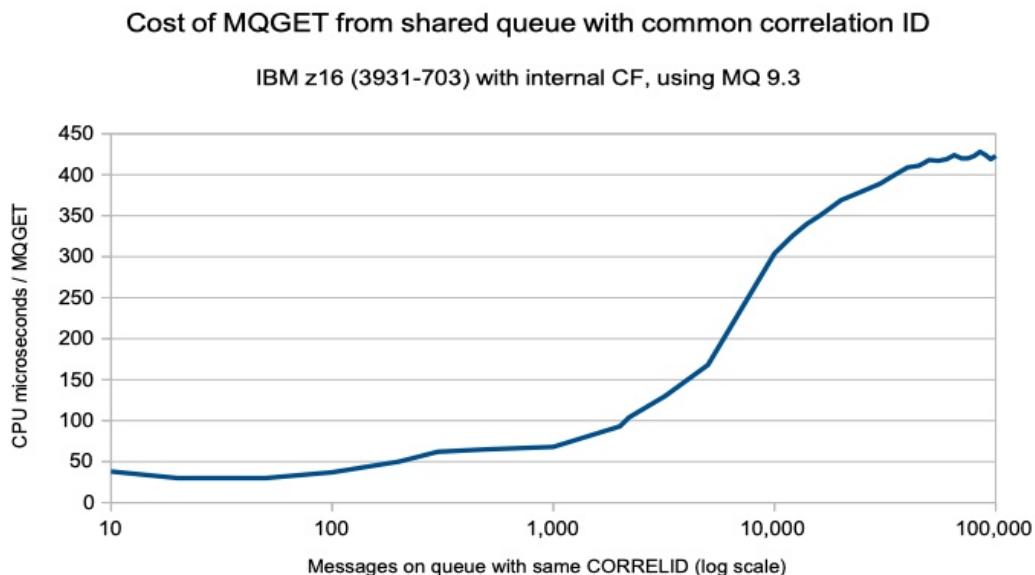
However, MQ shared queues are not optimised for MQGET when the queue is deep and a common key is used to identify many messages. As the number of messages using the common key increases, the cost increases as does the time taken to complete the MQGET.

If it is necessary to use a common value in the index of a shared queue, best MQGET performance will be achieved with less than 50 messages. Where more than 6,000 messages have the same value and an MQGET using the index is specified, the cost of the MQGET may be in excess of 9 times that of a get where only 1 message matches the search criteria.

Example: Consider a shared queue indexed by correlation ID that contains many messages and includes multiple messages with the same key.

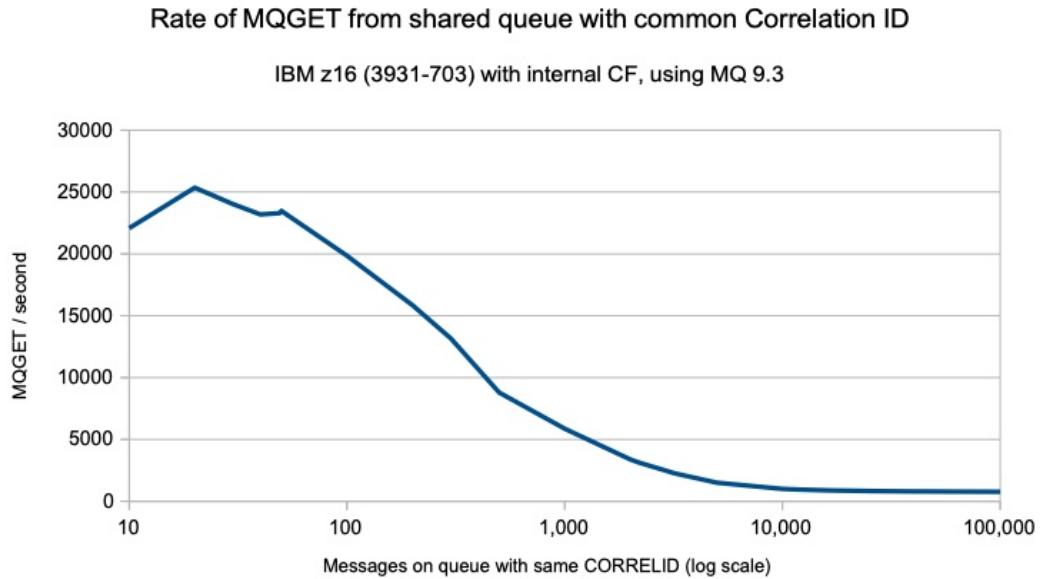
An application getting the messages is configured to only get the messages with the same known correlation ID.

The following charts demonstrate the cost of getting each message, as the number of messages with the same correlation ID increases.



Note: On IBM z16 with an internal CF, there is a noticeable increase in cost at approximately 50 and 1100 messages, and for each subsequent 1000 messages with the same correlation ID up to 5,200 messages with the same key. Once there are more than 25,000 messages with the same key, the costs are relatively static (within 10%).

The following chart demonstrates the impact of the increased cost on the rate that the application is able to get the matching messages.



Note: The measurements were run on a 3931-703 with an internal CF. With a remote CF, the rate of get may be significantly less.

In addition, depending on the number of matching messages, there may be re-drives to the Coupling Facility. In the worst case, where there are more than 5,200 messages with the same identifier, you will see 6 re-drives per MQGET.

The following data is an extract from the SMF 115 data containing the CF Manager data when putting 10,000 messages with a common CORRELID and subsequently getting those messages using the known CORRELID gives:

```
APPLICATION1, Structure-fulls 0
Single 30000, avg et in uS 5, Retries 0
Multiple 64235, avg et in uS 130, Retries 44235
Max entries 10034, Max elements 60064
```

Note that the number of retries is approximately 4.4 times the number of messages that were put. When there were more than 5,200 messages with the same CORRELID, each MQGET resulted in 6 re-drives. As less messages with the same correlation ID remain on the queue, the number of re-drives per message reduces.

Why is cost of MQGET higher when more than 5,200 messages have common identifier?

Once more than 5,200 messages are on a queue with a common identifier, the number of retries per MQGET for that specific identifier is restricted to 6, yet the MQGET cost has been shown to increase until stabilising when there 25,000 or more messages sharing the same MSGID or CORRELID.

The data explained in the [“MQ for z/OS - CF Statistics”](#) blog can help explain this.

For example, where the MQGET task attempts to get 2000 messages with a common CORRELID, MQ accounting data may show:

Get valid count	2000
Get valid destructive	2000
Get dest-specific	2000
CF time/verb	209 uS
CF Avg Sync elapsed time/verb	22 uS
CF Sync number of requests	4011
CF Avg Sync CF response time	11 uS
CF Avg Async elapsed time/verb	186 uS
CF Async number of requests	2834
CF Avg Async CF response time	131 uS
ReadList Avg Sync elapsed time/verb	17 uS
ReadList Sync number of requests	2011
ReadList Avg Sync CF response time	17 uS
ReadList Avg Async elapsed time/verb	186 uS
ReadList Async number of requests	2834
ReadList Avg Async CF response time	131 uS
Move Avg Sync elapsed time/verb	5 uS
Move Sync number of requests	2000
Move Avg Sync CF response time	5 uS

This data tells us that there were 2000 MQGETs, all of which were successful. As part of this there were 4011 synchronous CF requests and 2834 asynchronous requests, for a total of 6,845 CF requests.

Of these requests, 2000 were synchronous “move” requests (actually getting the message) and the remainder were “readlist” requests (identifying the message(s) to return), of which 2011 were synchronous and 2834 were asynchronous.

With regards to the “readlist” requests, the average asynchronous requests took 131 microseconds, compared to 17 microseconds for each of the synchronous requests.

As the number of messages sharing the common CORRELID increased, we observed both the response time for the **synchronous** requests increasing from 17 microseconds in the example to 150 microseconds, as well as the proportion of asynchronous “readlist” requests. This is despite the CF being very lightly utilised, i.e. this type of CF request is expensive compared to an MQGET of the next available message.

Frequent opening of un-defined queues

When an MQOPEN of a queue is requested, the queue manager will perform a look-up of local storage to get information about the queue. If successful, queue open processing will continue.

If the queue name is not found in local storage and the queue manager is part of a Queue Sharing Group, the queue manager will look for the queue in the DB2 table CSQ.OBJ_B_QUEUE. This check can be relatively expensive, of the order 8-10 times that of a successful queue open. There is cost incurred both by the queue manager in its' DB2 Server threads, plus in the DB2 master address space. The costs in the DB2 master address space is accounted as CPU time consumed by pre-emptible SRBs running on behalf of the master address space.

On our systems, 1 million attempts to open and close an undefined queue saw average costs of 59.4 CPU microseconds in the queue manager and 54.4 CPU microseconds in the DB2 address space.

As such, we would suggest that your applications do not attempt to open un-defined queues on a frequent basis as this may incur cost to your Db2 subsystem as well as your queue manager.

Example: When a queue manager is configured with class 3 accounting trace and an application attempts to open an MQ queue that has not been defined, an SMF 116 record may be written for the task.

An example of this scenario is shown:

```
VKW1 Batch Jobname:VWK1PUT Userid:SHARKEY
Start time Feb 22 07:10:24 2018 Started this interval
Interval Feb 22 07:10:24 2018 - Feb 22 07:10:25 2018 : 1.042905 seconds
Other reqs : Count 2
Other reqs : Avg elapsed time 201 uS
Other reqs : Avg CPU 31 uS
Other reqs : Total ET 0.000402 Seconds
Other reqs : Total CPU 0.000062 Seconds
== DB2 activity : 1 requests
> Average time per DB2 request-Server : 321 uS
> Average time per DB2 request-Thread : 321 uS
> Maximum time per DB2 request-Server : 321 uS
> Maximum time per DB2 request-Thread : 344 uS
> Bytes put to DB2 : 0
> Bytes read from DB2 : 0
```

In this example the DB2 request, with a response time of 321 CPU microseconds, can result in the equivalent CPU cost being shared between queue manager and Db2.

SHAREDYN Model Queues

In the case of using SHAREDYN model queues, you may see Db2 activity - potentially once to check for an existing queue and then again to create the queue definition. Subsequent uses of the resulting queue should not require accessing Db2.

Given we would not expect SHAREDYN queues to frequently be defined and deleted, we would expect the impact of SHAREDYN queue definition to be relatively small, however on our systems the cost of the define was of the order of 8 CPU milliseconds that was charged to the queue manager with additional cost to Db2.

Frequent opening of shared queues

With shared queue, there is a first-open and last-close effect which can increase the cost of opening a queue.

When an application opens a shared queue, if no other application on the queue manager already has the queue open, then the queue manager has to go to the coupling facility to register an interest in the queue and to get information about the queue. This is the first-open effect. If the queue manager already has the queue open, the information is already available and the queue manager does not need to register.

When an application closes a shared queue, if no other application on that queue manager has the queue open, then the queue manager has to go to the coupling facility to deregister an interest in the queue. This is the last-close effect.

Some applications naturally have the queue open for a long time, for example a channel, a trigger monitor, or long running applications. For high throughput, short-lived transactions, the queue may always have at least one application with the queue open. In these cases, the first-open and last-close effects are not seen.

If you have short running transactions with throughput such that the queue is not open all of the time, there may be little or no overlaps, which can result in each transaction experiencing first-open and last-close effects.

To compound this, if there are applications connected to different queue managers in the queue sharing group, that are opening and closing the queues frequently, the CF response time can be observed to increase due to each queue manager to attempting to register interest queue.

To further compound this, the type of open option can affect the first-open and last-close effect.

- With MQOO_INPUT_SHARED, the impact of the lock prior to registration increases exponentially as the number of queue managers attempting to obtain the lock increases. These registrations will also present themselves in the form of increased latch times, specifically DMCSEGAL (latch 11).
- Using MQOO_OUTPUT with frequent opening and closing of a shared queue, particularly when as the number of queue managers attempting to register/de-register the usage of the queue increases, can result in increased CPU cost by the MQOPEN API, which manifests as additional CPU usage in the application address space.

MQOO_INPUT_SHARED Example: In the following scenario we define 3 queue managers in a QSG, where each queue manager is running on a separate LPAR.

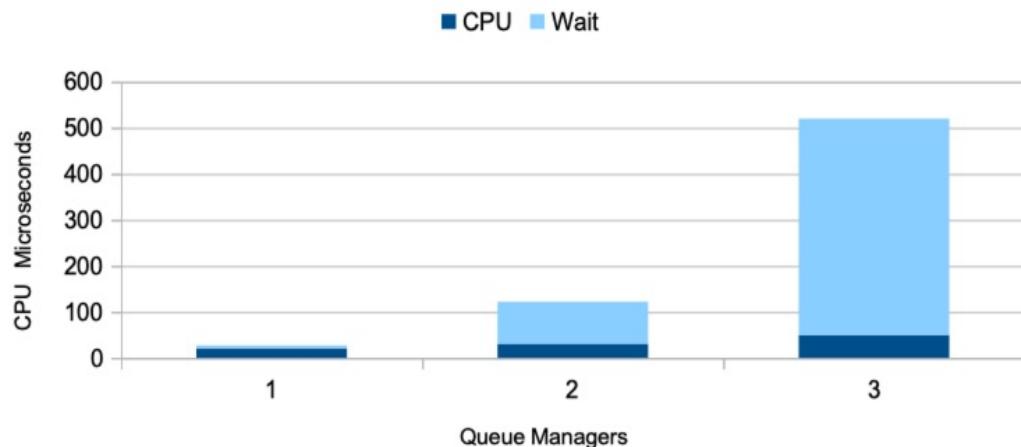
A single application is connected to queue manager 1 and performs 1 million MQOPEN with option MQOO_INPUT_SHARED and MQCLOSE calls of a single common shared queue.

The measurement is then repeated with the same application running against queue managers 1 and 2, and then finally against queue manager 1, 2 and 3.

The cost shown is that of the average of the 1 million MQOPENS on queue manager 1 using class 3 accounting data.

MQOPEN of Shared Queue with increasing queue managers

Queue opened with: MQOO_INPUT_SHARED



The costs are shown in the following table for clarity:

Queue Managers	1	2	3
MQOPEN CPU	21	31	51
MQOPEN Elapsed	27	122	520
Wait (Elapsed - CPU)	6	91	469

The CPU cost of the MQOPEN on queue manager 1, increased from 21 CPU microseconds by 47% when applications on 2 queue managers attempted to open the same queue. This further increased another 61% when the application on the third queue manager was introduced.

As the table indicates, the "wait" time shown in the chart is calculated from the elapsed time minus the CPU time.

The wait time increases from 6 by 85 microseconds when running the workload against 2 queue managers, and a further 378 microseconds with 3 queue managers.

This increase in wait time is largely due to waiting for lock.

Additionally, we saw increased DMCSEGAL (latch 11) times, for example in these measurements the time spent waiting for DMCSEGAL increased from 7 to 16 microseconds per MQOPEN with increased queue managers. With this latch being held across the CF access, a less responsive CF may result in increased DMCSEGAL times.

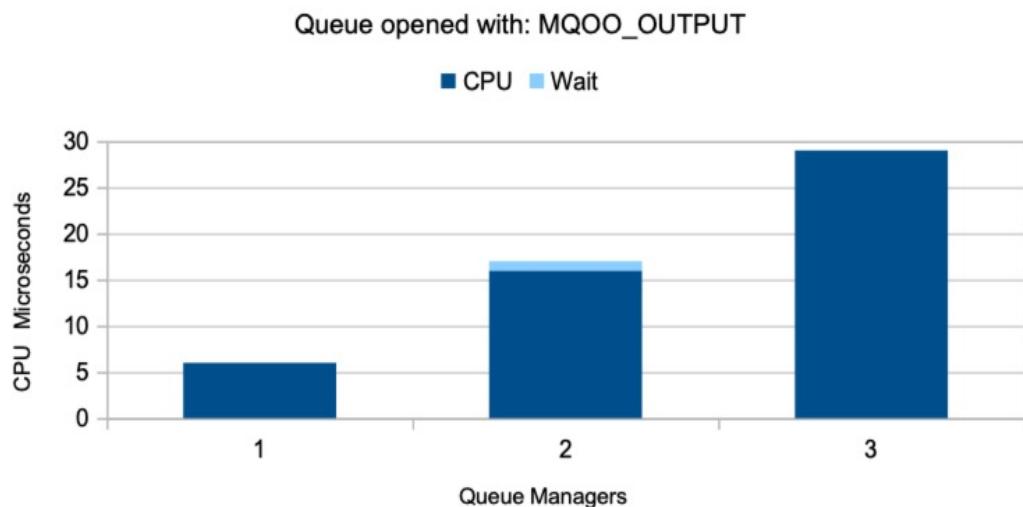
MQOO_OUTPUT Example: In the following scenario we again define 3 queue managers in a QSG, where each queue manager is running on a separate LPAR.

A single application is connected to queue manager 1 and performs 1 million MQOPEN with option MQOO_OUTPUT and MQCLOSE calls of a single common shared queue.

The measurement is then repeated with the same application running against queue managers 1 and 2, and then finally against queue manager 1, 2 and 3.

The cost shown is that of the average of the 1 million MQOPENs on queue manager 1 using class 3 accounting data.

MQOPEN of Shared Queue with increasing queue managers



The costs are shown in the following table for clarity:

Queue Managers	1	2	3
MQOPEN CPU	6	16	29
MQOPEN Elapsed	6	17	29
Wait (Elapsed - CPU)	0	1	0

The CPU cost of the MQOPEN on queue manager 1, increased from 6 CPU microseconds by 10 CPU microseconds when applications on 2 queue managers attempted to open the same queue. This further increased another 13 CPU microseconds when the application on the third queue manager was introduced.

As the table indicates, the "wait" time shown in the chart is calculated from the elapsed time minus the CPU time.

In the case of MQOO_OUTPUT, there was minimal wait time as additional queue managers were added, and this was reflected with no increase in latch times for the MQOPEN.

Whilst opening the shared queue for output saw minimal wait time, the increased CPU cost also resulted in additional load on the CF due to the queue manager attempting register the queue usage. In some circumstances, particularly where many queue managers were attempting to register access to the same shared queue, the MQOPEN can see "CF retries". This can be seen in the MQSMF's task report (available as part the Performance SupportPac [MP1B](#) "Interpreting accounting and statistics data"), for example:

Open count	565020	SIXC01
Open avg elapsed time	17 uS	SIXC01
Open avg CPU time	16 uS	SIXC01
Open CF access	565020	SIXC01
Open no CF access	0	SIXC01
CF time/verb	11 uS	
CF Avg Sync elapsed time/verb	11 uS	
CF Sync number of request	1591903	
CF Avg Sync CF response time	4 uS	
CF Retries	337039 out of 1591903 (21.2%)	

In this example, there were 337039 CF requests that had to be re-tried before the queue manager was able to successfully register access to the SIXC01 shared queue.

Note that these are CF retries and not CF re-drives. CF re-drives tend to occur when the buffer passed to the CF is not sufficiently large for the returning data, whereas a CF retry may occur when the MQ queue manager is unable to complete its request due to another MQ queue manager performing a similar request.

How can I tell if I am seeing first-open or last-close effects?

Performance SupportPac [MP1B](#) “Interpreting accounting and statistics data” provides a task report that shows information like:

```
VKW1 Batch Jobname:VKW1PUT Userid:SHARKEY
Open CF access      73490
Open No CF access   0

Close CF access     73490
Close No CF access  0
```

These fields show that there were first-open and last-close effects, with each open and close resulting in CF access.

If there is contention occurring due to other queue managers attempting to access the same MQ resources, particularly when opening queues for output, there may be information like:

```
CF Retries          337039 out of 1591903 ( 21.2%)
```

In our measurements, CF retries were more frequent when opening the queues for output.

Can I reduce the impact from locking on my shared queues?

If you see a large number of first-open or last-closes, including from CF retries, then using an application connected to each queue manager that opens the shared queue(s) and then goes into a long sleep, can significantly reduce the time and cost of the frequent MQOPEN in your applications.

Note: In order to minimise the impact of first-open and last-close, the application connecting must open the shared queue(s) using the same type of open as the applications performing the frequent queue opens.

That is to say, if the frequently opened queue is opened for output, then the application used to mitigate the effects of first-open and last-close should also open the queue for output.

Similarly if the frequently opened queue is open for input, then the application should also open the queue for input.

If the application used to mitigate the first-open effect uses a different open-type, there will be no benefit as the frequent opening task will still need to access the CF.

Using this long running application means that CF access is no longer required by the application that is opening and closing the queues frequently, which significantly reduces cost and response time.

Example: Repeating the earlier measurements with an additional application connected to each queue manager that MQOPENed the common shared queue and then went into a long sleep, the average cost of the 1 million MQOPENS dropped to 1 CPU microseconds regardless of the number of queue managers.

The following table shows the total system cost when running the application to open / close a common shared queue 1 million times using MQOO_INPUT_SHARED. Cost is CPU seconds for LPAR 1.

Queue Managers	1	2	3
Baseline	63	137	180
With application holding the queue open	14	15	14

This table shows that the cost of the workload running the MQOPEN and MQCLOSE was:

- Significantly reduced when the queue was already open
- Did not increase in cost as more applications and queue managers performed the same workload.

Is using an application to hold the queue open always appropriate?

In short, not all shared queue configurations will benefit from using an application that opens the shared queue with the appropriate open option and then sleeps.

For example, if the shared queue is configured with TRIGTYPE(FIRST), then such an application can prevent the triggering mechanism from initiating the triggered application.

It is worth noting that TRIGTYPE(EVERY) can benefit from the long running application - and in a simple CICS environment, we measured a reduction in MQOPEN costs of up to 80% and MQCLOSE costs reduced by up to 90%.

Using GROUPID with shared queues

Message grouping allows logical groups of messages to be associated together.

Comparing performance of GROUPID with CORRELID

Consider whether using a queue indexed by GROUPID is appropriate. The performance of a workload relying on GROUPID where the group size is 1 message, is worse than the equivalent performance of CORRELID or MSGID, particularly with shared queues.

The following charts compare a workload where a message is randomly selected from a pre-loaded queue with a fixed number of messages each with a unique identifier, where the unique identifier is either a unique group or correlation ID. The selected message is then gotten, processed and replaced on the queue. For simplicity and comparison purposes, the measurements using GROUPID use a group consisting of 1 message.

When using the group identifier the application specified:

- MQGMO MatchOptions include MQMO_MATCH_GROUP_ID
- MQGMO Options include MQGMO_ALL_MSGS_AVAILABLE and MQGMO_LOGICAL_ORDER

Chart: Compare transaction rate when using shared queue indexed by CorrelID and GroupID:

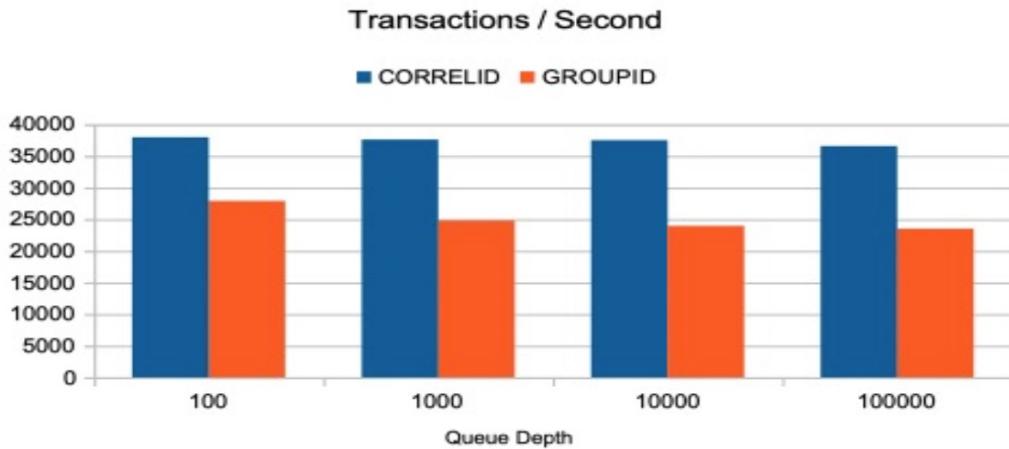
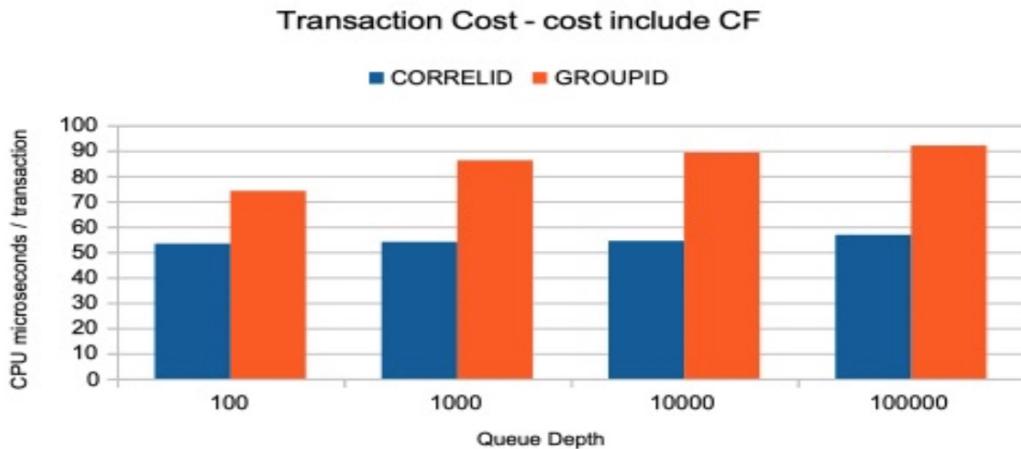


Chart: Compare transaction cost when using shared queue indexed by CorrelID and GroupID:



What these charts show is that using GroupID with a group size of 1 is more expensive than CorrelID, with the additional cost in the MQGET processing. This additional cost directly impacts the MQGET rate.

The depth of the queue also has a larger impact on the workload using GroupID than CorrelID.

Comparing performance of GMO options

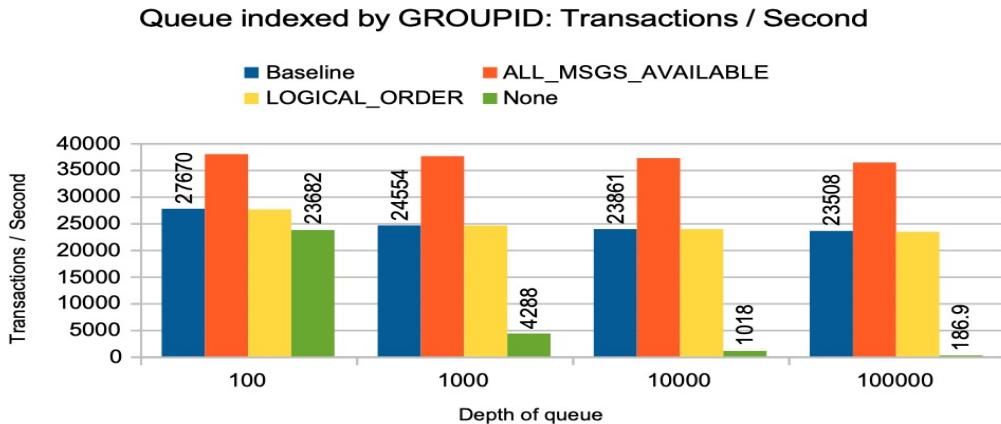
The performance of message grouping can be significantly affected by the MQGMO options specified, particularly with volatile queue depths.

In the previous comparison, both MQGMO_ALL_MSGS_AVAILABLE and MQGMO_LOGICAL_ORDER were specified, but these are not enforced. However, despite not being enforced, the performance characteristics can be significantly different.

The measurements shown in this section demonstrate the effect of different combinations of MQGMO options when processing messages in groups of 1 message.

Configuration	ALL_MSGS_AVAILABLE	LOGICAL_ORDER
Baseline	Yes	Yes
ALL_MSGS_AVAILABLE	Yes	No
LOGICAL_ORDER	No	Yes
None	No	No

Chart: Impact of MQGMO options on transaction rate:



Note: There is a significant drop in performance in the “none” configuration as the queue depth increases.

In the first 3 configurations, MQ’s class(3) accounting data shows that the MQGETs are resolved as get-specific.

The “none” configuration results in the MQGET of the specific GroupID being processed as get-next, i.e. MQ resorts to scanning the queue for the matching GroupID.

Furthermore, the “none” configuration with a single application task caused 1 CF processor to be run at just under 100%. Additionally there was an increasing proportion of asynchronous CF requests, with the response times increasing from 4.3 microseconds (sync) to 500 microseconds (async) as the depth of the queue increased.

When running with 4 tasks and their own set of message groups on the same queue, the 4 processor CF showed utilisation at 92.6% which is far higher than we would typically recommend. There was also a decrease in overall throughput despite having 4 times the number of applications running.

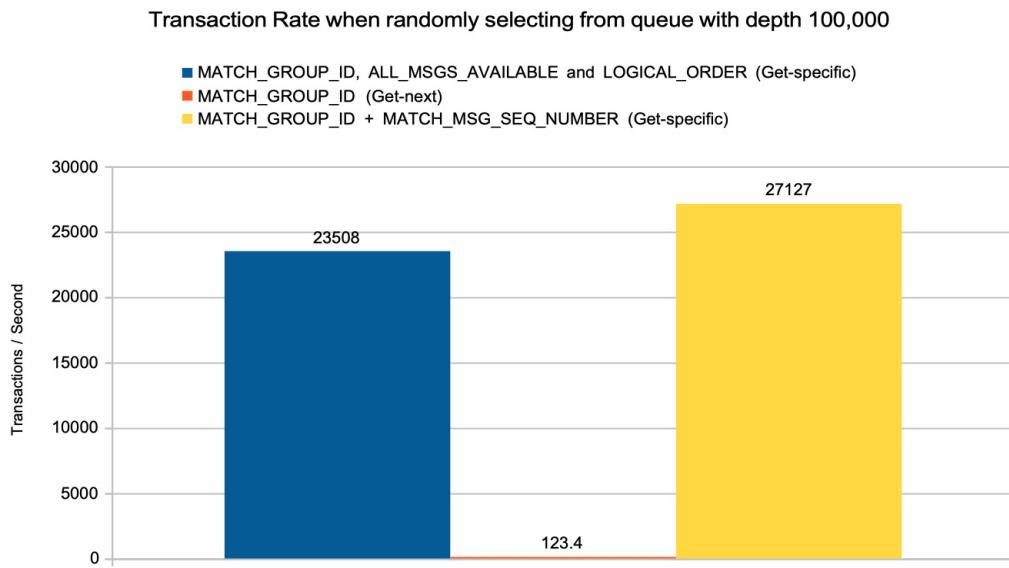
Avoiding Get-Next when specifying GroupID

There are many ways to get a group of messages, but some are more optimal than others - particularly when using shared queues.

For a known GroupID of messages on shared queues, there are 2 better performing MQGMO options combinations:

1. MatchOptions including MQMO_MATCH_GROUP_ID plus Options including MQGMO_ALL_MSGS_AVAILABLE and MQGMO_LOGICAL_ORDER.
2. If MQGMO_LOGICAL_ORDER is not appropriate,
MatchOptions include MQMO_MATCH_GROUP_ID and MQMO_MATCH_SEQ_NUMBER and specify MQMDE_MsgSeqNumber (default is 1).
 - This will drive MQGET-specific processing, whereas if the MQMO_MATCH_SEQ_NUMBER is not specified, get-next processing will be used. Get-next processing, particularly when using GroupID with shared queues can be costly in terms of CF usage in the event of deep queues or many applications.

Chart: Impact of Get-next when specifying GroupID on deep queues:



Note: If you are using GroupID with shared queues, check the class(3) accounting data to see if the application is using get-specific or get-next.

Using GroupID from a JMS application will mean the access will use get-next processing, which as demonstrated earlier may see poor performance when the queue depth increases.

Private queues will use the GroupID index even when the MQMO_MATCH_MSG_SEQ_NUMBER is not specified, which result in get-specific processing occurring. Shared queue is unable to mirror this behaviour due to indexing limits on shared queue list structures.

Using Message Selectors

IBM MQ supports the use of message selectors to identify messages by specific properties. Generally this is a relatively expensive process as the queue manager must scan each message to determine if the current message has matching properties.

When the queue contains many messages to scan, the cost can be prohibitive.

A complex message selection criteria also may be expensive.

Who pays for the cost of message selection?

For **private queues** the cost will be incurred by the application, or if using client connections, this cost will be charged to the channel initiator adaptor task.

For **shared queues** the cost will be incurred by both the application, or if using client connections the channel initiator adaptor task, and the Coupling Facility.

Is there a good message selector to use?

The optimal message selectors to use are “select by correlation ID” and “select by message ID”, particularly with the appropriate INDXTYPE defined.

In our measurements using “select by correlation ID” performs similarly to “get by correlation ID”.

“Select by correlation ID” requires the format of the selector to be:

```
JMSCorrelationID='ID:<lower case, hexadecimal correlation ID>'
```

How do I know if I am using a good message selector?

MQ does not offer any statistics to specifically show when message selectors are being used to identify messages.

The use of class(3) accounting data for the task can indicate whether the MQGET is get-specific and potentially able to use the index, or is scanning the queue.

If the application is scanning a private queue, you may see high numbers of “skips” in the task record, particularly when accessing queues with many messages, e.g.

-MQ call-	N	ET	CT	Susp	LOGW	PSET	Epages	skip
Get : 541	43822	43610	0	0	0	0	0	48363579

By dividing the number of skipped pages by the number of messages, we can calculate the average number of skips per MQGET.

In the above example, we are seeing an average of 43,610 skips per MQGET, where each MQGET costs an average of 43.6 CPU milliseconds. Given this is a 2KB message, we would regard this as an expensive MQGET.

In this example, the number of skipped message, particularly relative to each MQGET, causes this high cost.

Also note that there is little difference between the average elapsed and CPU time. In this particular interval the task used 23.7 seconds in MQGET (elapsed) of which 23.5 seconds were used in CPU.

Message selector performance

In this section we look at the performance of message selectors where there are increasing numbers of messages to be checked for the desired message property.

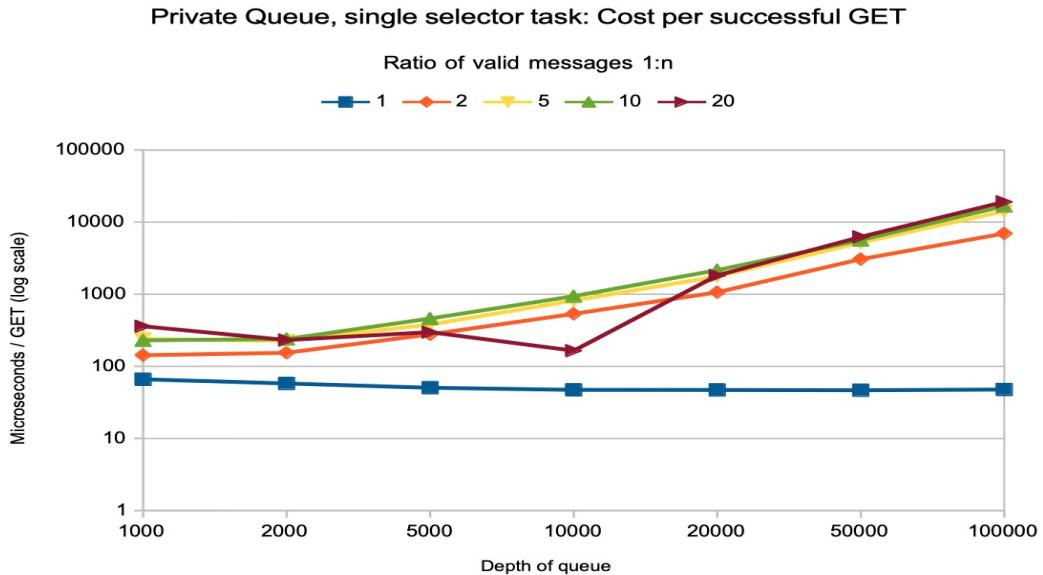
- The queue manager is configured with sufficiently sized buffer pools such that any workload is not affected by page set I/O.
- The queue manager is configured with a sufficiently sized application structure such that the messages are not offloaded to SMDS or Storage Class Memory.
- The tests are configured with a fixed number of non-persistent messages on a single queue, ranging from 1,000 to 100,000 messages.
- All messages have 10 properties and a 2KB payload.
- A varying percentage of messages are “valid”, i.e. we get all messages with a property of “colour=red” and there may be between 0 to 95% of messages that do not contain a matching property. This is based on the “colour=red” messages forming the following ratio of the messages on the queue:
 - 1:1 - all messages are red.
 - 1:2 - 50% of messages are red.
 - 1:5 - 20% of messages are red.
 - 1:10 - 10% of messages are red.
 - 1:20 - 5% of messages are red.
- The application runs as a single threaded client, using a SVRCONN channel to get the MQ messages and uses the following logic:
 - Connect.
 - Specify selection of all red messages as part of MQOPEN.
 - MQGET until the desired number of messages are retrieved
 - MQCLOSE.
 - Disconnect.

Message selector performance with private queues

The following chart shows the average cost of a successful MQGET.

When the ratio is 1:1, i.e. all messages are valid, the depth of the queue does not detrimentally impact the cost of the MQGET. As the proportion of valid messages decrease, the queue manager needs to skip more messages to find the next valid message, which results in increased cost.

Chart: MQGET cost using message selectors on private queues with increasing depth:



Note: The lower cost observed with queue depth of 10,000 and a ratio of 1 in 20 valid messages occurred as the valid messages were grouped together near the head of the queue. As such, there were less skips required to find all of the desired messages.

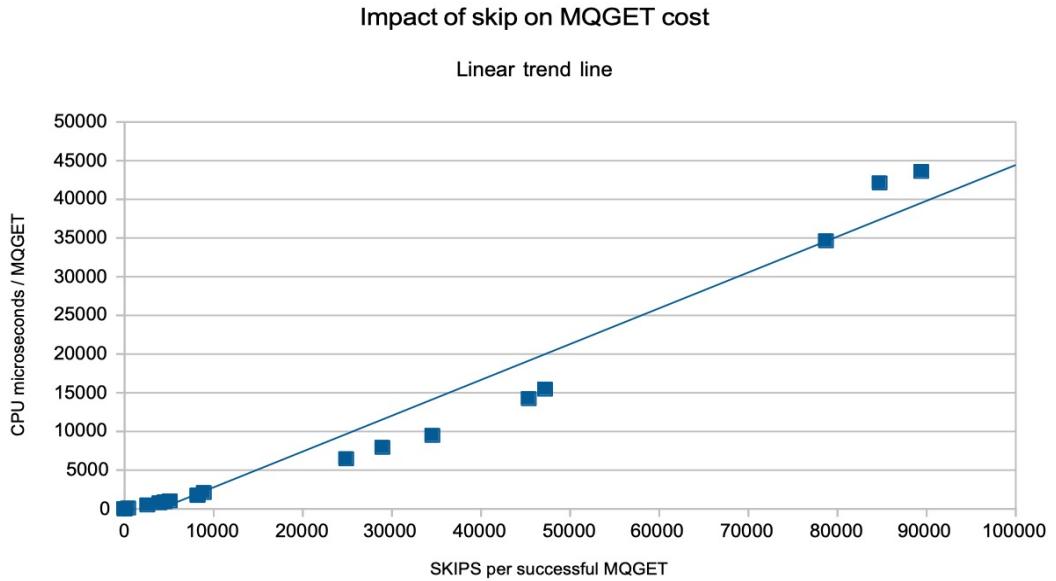
There is little that can be done to influence the order of the messages on the queue such that the number of “skips” is reduced, other than using the correlation ID to hold a property and the use of “select by correlation ID”.

As mentioned previously, the number of skipped messages has a direct impact on the cost of the MQGET when using message selectors on private queues.

When only a proportion of messages match the selection criteria and the depth of the queue increases, MQ must scan (or skip) more pages to find the desired message.

Skips can be seen in the accounting class(3) data for the task running the message selection, which would be a channel initiator adaptor task for a client connection.

Chart: Plotting number of skips against MQGET cost:



The relationship between cost and number of “skips” is linear - and for best performance it is preferable to minimise the number of messages skipped.

The increasing cost also directly affects the achievable transaction rate. If an MQGET takes 20 microseconds, theoretically the application could process 50,000 MQGETs per elapsed second.

If that MQGET takes longer, for example a message that required 2500 “skips” cost 500 microseconds in our measurements, would have a theoretical maximum MQGET rate of 200 per elapsed second. Using additional tasks using MQGET would not necessarily improve the performance as each application is already using 1 processor at 100% utilisation.

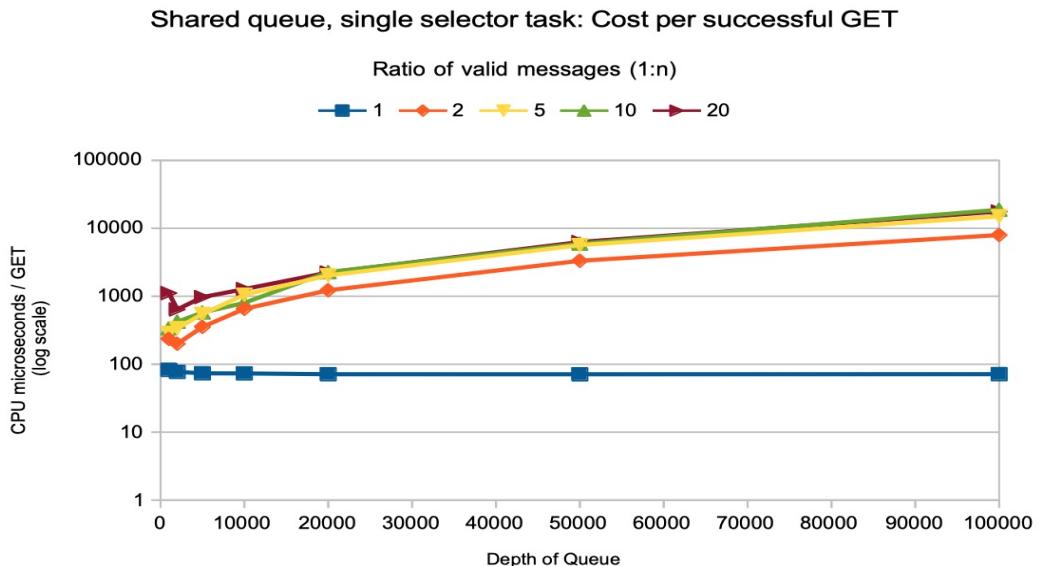
Message selector performance with shared queues

When using message selectors against messages on shared queues, MQ must retrieve each message from the Coupling Facility, and potentially SMDS or Db2, scan the message, parsing for the desired message property. This can be a CPU intensive request.

- The first request will start at the head of the queue and MQ will scan down the queue until a match is found.
- Subsequent requests, within an MQOPEN handle, will typically scan from the next message.
- However if messages continue to arrive, the cursor may be reset to the head of the queue.
- Similarly if the queue is closed and then re-opened between requests, the cursor will reset to the head of the queue.

As with private queues, when the ratio is 1:1, i.e. all messages are valid, the depth of the queue does not detrimentally impact the cost of the MQGET. As the proportion of valid messages decrease, the queue manager needs to retrieve and scan more messages to find the next valid message, which results in increased cost.

Chart: MQGET cost using message selectors on shared queues with increasing depth:



Note: When the depth of the queue is low, the sample size of MQGETs is relatively low which accounts for the slightly higher average cost. As the sample size grows the average cost initially drops and then increases as the effect of the queue depth becomes more of a factor.

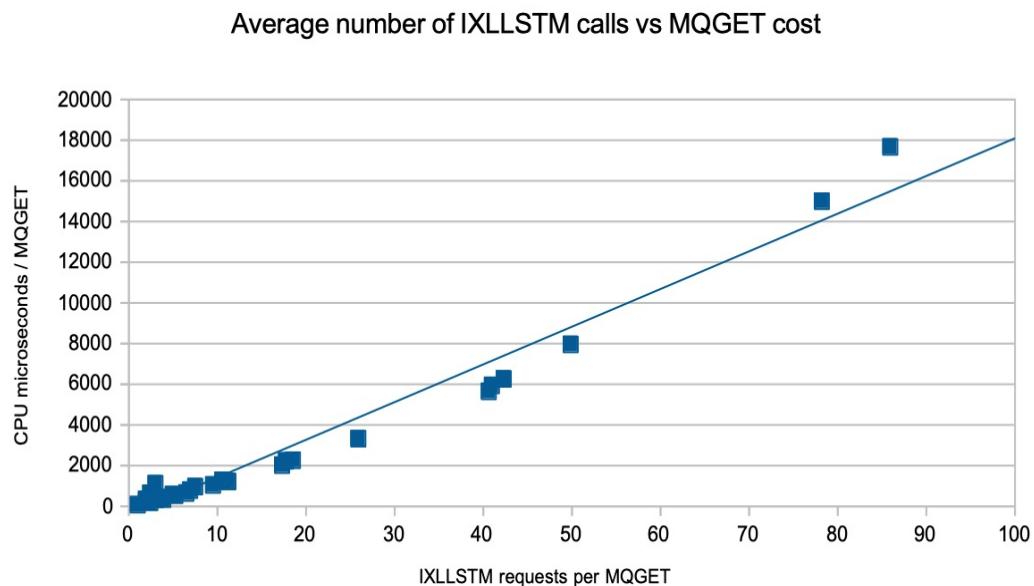
When only a proportion of the messages on the queue match the selection criteria, MQ must retrieve and scan the messages to determine if the current message is a match. As with private queue, the number of messages scanned and not matching, affects the cost of the successful MQGET.

Unlike the private queue configuration, there is not a specific “skipped” attribute reported but there are alternatives:

- The class(3) accounting data has an attribute WTASCMEC, which counts the number of IXLLSTM calls made by the task. This module is used to retrieve multiple messages in a single call, and can be used to give an indication of the number of messages scanned.

- The RMF™ Coupling Facility Activity report can indicate how many CF requests occurred for a particular structure. If the selector applications are the only users of the structure, this can be used to determine the ratio of CF requests to MQGETs.

Chart: Plotting number of IXLLSTM calls per MQGET against MQGET cost:



Using selectors on shared queue messages will result in increased CF usage. In our measurements, the channel initiator adaptor task was typically 100% busy based on the elapsed time of which 50-90% was CPU. The difference between CPU and elapsed time was due to CF access times. Our CF is local and highly responsive, yet even with a single selector task, the CF was driven at the equivalent of 1 CF processor at 40% busy.

Checklist: Using client-based selectors

- Selecting on message properties is more expensive than using Message ID or Correlation ID.
- Selecting using one of the optimised message selectors, for example JMSCorrelationID is more efficient on appropriately indexed queues, i.e. INDXTYPE(CORRELID).
- Complexity of selection criteria may add cost to identifying the appropriate message.
- Queue depth, as well as where the desired message is on the queue, is a significant factor in the cost and time taken to identify the desired message.
- Ensure there are sufficient CHIADAP (channel initiator adaptor) tasks available such that at least 1 is unused even at peak times, to avoid waiting for an adaptor.
 - An expensive MQGET request from a client, such as one using message selection will block its' adaptor task
 - When using message selection on shared queues, the adaptor task in use may see significantly longer elapsed time than CPU time due to the time spent accessing data in the CF. Waiting for the CF response still blocks the adaptor task.
- The cost of the MQGET can be affected by the number of messages that have to be scanned before finding the desired message.
 - Private queue - check accounting class(3) data for the task, specifically look at the number of skips required to satisfy all the MQGETs in the interval. The number of skips can have a direct impact on the cost.
 - Shared queues - check accounting class(3) data for the task, specifically looking at the WTASCMEC variable (number of IXLLSTM calls) and calculate the ratio of calls to MQGETs. A high ratio of WTASCMEC to MQGETs may indicate many messages having to be retrieved and scanned.
 - Alternative for shared queues - if the only queues in the application structure are accessed in the same way, the RMF™ Coupling Facility Activity report can help indicate whether the number of CF requests corresponds to the number of messages gotten. Where there are many requests to MQGETs, the queue manager is likely having to scan many messages to find a match.
 - Consider separate queues if frequently selecting particular properties, or store the message property in the Correlation ID.
 - MQ is not a database and as such database-type queries on message properties is not a particularly efficient method to identify a desired message.
- Adding more selector tasks may make performance worse. There will be additional load on the CPU, and if accessing shared queues there may be higher CPU usage and increased serialisation on the CF resources, which may affect the performance of the existing tasks. Additional client-based selectors will also put further load on the channel initiator adaptor tasks.
- Ensure the MQOPEN handle is retained as long as possible to get all messages, which may avoid the cursor resetting to the head of the queue.

Temporary Dynamic (TEMPDYN) Queues

A temporary dynamic (TEMPDYN) queue is created when an application issues an MQOPEN API call with the name of the model queue specified in the object descriptor (MQOD).

The defined queue is created in the STGCLASS as specified in the model queue definition, which will ensure the TEMPDYN queue is defined in the associated page set. The MQSC command “DEFINE PSID” command will determine which buffer pool is used to host the messages stored on the queue.

Note: The buffer pool used by the TEMPDYN queue may be reported inaccurately by Accounting class(3) data.

The TEMPDYN queue will be deleted when the application that created the queue issues an MQCLOSE against the queue. For a TEMPDYN queue, MQCO_NONE, MQCO_DELETE and MQCO_DELETE_PURGE will all have the effect of deleting the queue and purging any messages on it.

If the TEMPDYN queue is empty or has only committed messages, the messages will be purged and queue will be deleted synchronously.

Should there be uncommitted messages, MQ will purge the messages and delete the queue asynchronously.

TEMPDYN queues - MQOPEN

Opening a temporary dynamic queue is typically slower and more expensive than opening a pre-existing queue.

The following is an extract of Accounting class(3) data for an application that opens a temporary dynamic queue and a pre-existing queue.

Queue Type	Elapsed	CPU	Suspended
Pre-existing queue	2	2	0
Temporary dynamic queue	225	14	211

In this example the additional 211 CPU microseconds of “suspend” time is largely spent in a single log I/O request as MQ must harden the queue definition - even though it is temporary.

TEMPDYN queues - MQCLOSE

When the MQCLOSE is issued against the created TEMPDYN queue, MQ will attempt to purge any messages and delete the queue. As mentioned earlier, provided any messages on the queue have been committed, the messages and queue can be deleted synchronously.

Should there be any uncommitted messages on the queue, MQ will purge the messages and delete the queue at a later stage via the use of an asynchronous task.

When running with a high turnover of temporary dynamic queues that are reliant on asynchronous processing due to uncommitted messages, there can be a backlog of work for the asynchronous task to complete. If there is sufficient backlog, the buffer pool may be insufficiently sized for the running workload and I/O to page set may occur.

One method to help determine whether the asynchronous task is performing a large number of purge/deletes is to review the output from the MQSMF application, part of performance report [MP1B](#) “Interpreting accounting and statistics data”, that generates the Data Manager report in the DATA DD statement, e.g.

MVAA,VTS1,2020/02/11,10:50:40,VRM:914,
Obj Cre [130656](#), Obj Puts 0, Obj Dels [130655](#), Obj Gets 130657

If the number of object deletes is similar to the number of object creates, the queues are mostly deleted synchronously.

If the number of object deletes is approximately twice that of object creates, the queues are deleted asynchronously.

To ensure the messages are purged and the queues are deleted synchronously, ensure the messages are committed prior to the MQCLOSE API request. This can be achieved using MQCMIT in batch or, if running in a CICS environment via the use of EXEC CICS SYNCPOINT.

Chapter 7

Queue Information

Tuning queues

There are a number of options that can be applied to queues that can reduce cost or help identify problems in the appropriate environment.

Queue option ACCTQ

Specifies whether accounting data collection is to be enabled for this queue. In order for this data to be collected, it is necessary to enable class 3 accounting data. Whilst setting the ACCTQ attribute to a value other than QMGR can allow targeted analysis of costs, there is a risk that when evaluating the data gathered, there will be incomplete data from access to queues where the ACCTQ queue attribute is set to a different value.

Queue option DEFRESP

Specifies the behaviour to be used by applications when the put response type, within the MQPMO options, is set to MQPMO_RESPONSE_AS_Q_DEF.

Using asynchronous puts can improve message put rate and reduce the round-trip time when putting messages using an MQ client application. If your messaging requirements allow it, running with asynchronous puts from a client application can reduce the cost of the put on the z/OS queue managers and channel initiator by between 18% and 55% for messages ranging in sizes 100,000 to 1000 bytes.

Queue option DEFREADA

Specifies the default read ahead for non-persistent messages delivered to the client. Enabling read ahead can improve the performance of client application consuming non-persistent messages. In a queuing model, where the client application is just getting messages from a queue and not putting messages, we have seen the get rate improve 4 times over synchronous gets and costs drop by between 12 and 55%.

High-latency networks, where the time to respond to a request is extended due to a long time in the network can see significant improvements in MQGET – however a request/reply model will not benefit in this way.

Queue option MONQ

Controls the collection of online monitoring data for queues and is supported on local and model

queues only.

Queue option PROPCTL

This attribute specifies how message properties are handled when the messages are retrieved from the queue. Specifying a value of V6COMPAT can reduce the amount of parsing the queue manager has to complete and can reduce the cost.

Maximum throughput using non-persistent messages

What factors affect non persistent throughput

Throughput for non-persistent messages in private queues:

- Is ultimately limited by CPU power assuming messages can be contained within a buffer pool without spilling to DASD page sets.
- Where messages do have to be read from page sets then the I/O rate sustainable to that DASD will be the constraining factor for MQGET.
- In practise, the limiting factor for non-persistent throughput is likely to be in business logic IO rather than anything internal to IBM MQ.

Throughput for non-persistent messages in shared queues depends on:

- For messages up to 63KB (64512 bytes)
 - z/OS heuristics which can change CF calls from synchronous to asynchronous.
 - The type of link(s) between individual z/OS's and the CF(s).
 - This affects the elapsed time to complete CF calls and so influences the heuristics.
 - The CF machine type and CF CPU %BUSY.
- For messages > 63KB
 - As above for up to 63KB messages plus, the throughput performance of the Shared Message Data Set (SMDS) or the DB2 data sharing group tablespace used to store these messages.

Private queue

What is the maximum message rate through a single *private* queue ?

Using a 3 processor MVS image on a 3931-7K0 system running z/OS 2.5 and IBM MQ for z/OS 9.3 we could sustain the following non-persistent message rates to a private queue.

MQ for z/OS 9.3 on 8561-7G1	
Message size	Message rate / sec
1,000	114,060
5,000	107,230
10,000	102,600
30,000	82,720
100,000	49,520

Sustained means, in this case, that messages are MQPUT/MQCMIT and MQGET/MQCMIT at about the same rate so that the queue does not get very large.

We run four MQPUT/MQCMIT jobs in parallel with a four MQGET/MQCMIT jobs to obtain these results. Each job has 400,000 iterations.

NOTE: When running in a WLM controlled environment, it is advisable to ensure that the getting application has equal or higher priority than the putting application otherwise there is a risk that the queue depth will increase, resulting in page set I/O as bufferpools fill.

Throughput for request/reply pairs of private queues

The following message rates are for locally driven request/reply scenarios.

Each request/reply scenario uses:

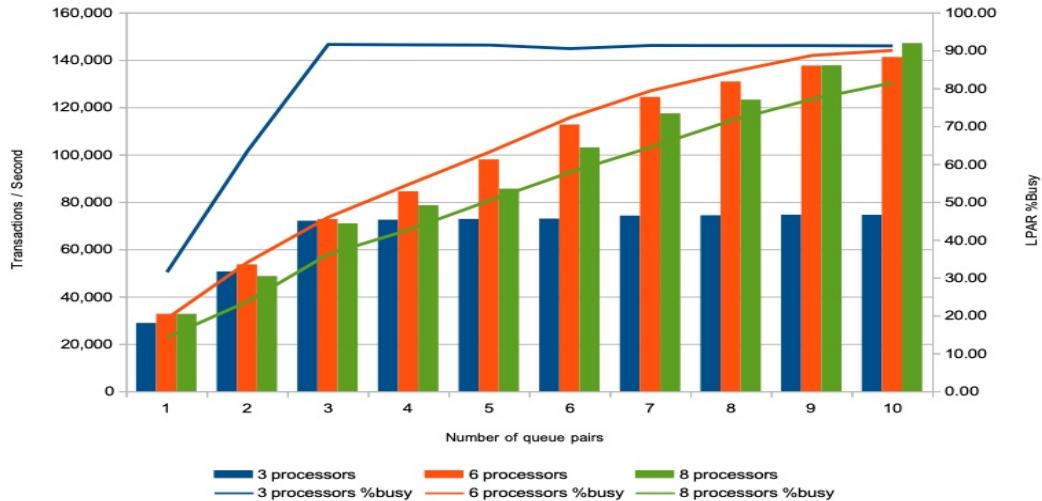
- One or more request/reply applications, each of which uses
 - A pair of queues (a common server queue and a common ‘indexed by MsgId’ reply queue)
 - One or more reply programs using that pair of queues using MQGET, MQPUT, MQCMIT
 - One or more requester programs per reply program using out-of-syncpoint MQPUT and MQGET by MsgId for the reply message specific to that requester.
 - Queue depths were always low, as is usual for request/reply applications. Thus no page set IO is required.

Locally driven Request/Reply on 3931-7K0 using 1000 byte non-persistent messages. Measurements on 6-way LPAR unless otherwise stated.					
Q pairs	Repliers / Q pair	Requesters / Q pair	Msgs/sec	Txns/sec	CPU Microsec / transaction
1	1	1	66664	33332	34
1	4	1	65196	32598	35
1	4	2	107973	53986	41
1	4	3	134330	67165	41
1	4	4	153845	76923	44
1	1	4	97295	48647	34
2	2	4	134528	67264	37
3	3	4	156725	78362	41
4	4	4	158538	79269	42
Following measurements are on 3-way LPAR					
1	1	1	57691	28846	33
2	1	1	101227	50613	37
3	1	1	144079	72039	38
4	1	1	144970	72845	38

The following chart shows how running non-persistent workload with increased numbers of processors can improve the throughput.

Running 1 requester/server pair for each queue pair, increasing queue pairs

Comparing throughput with varying numbers of processors. Transactions / second and %busy, using IBM MQ 9.3

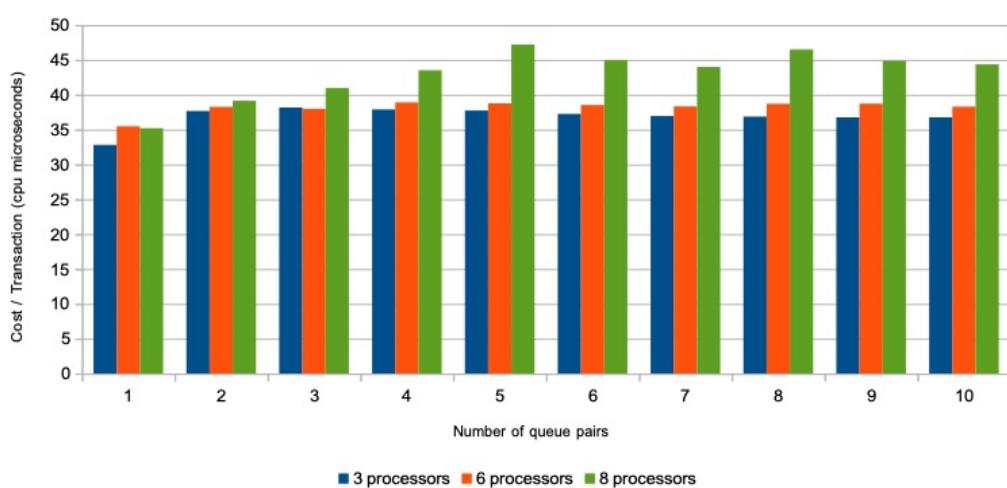


NOTE: When there are 3 processors available, the rate at which the throughput peaks corresponds directly with the number of processors available on the above chart. As the number of processors equals or exceeds 6, the transaction rates still increase accordingly but the peak rate is less marked.

Transaction cost can be affected by the number of processors available. The following chart compares the transaction costs observed for the previous measurement.

Running 1 requester / server pair for each queue pair, increasing queue pairs

Comparing transaction cost with varying numbers of processors



It may be worth noting that the 8 processor configuration transaction costs are higher than when similarly configired measurements on IBM z15. This is because on the IBM z15, the 8 processors were allocated on a single chip, whereas on IBM z16 we are seeing a maximum of 6 processors per chip. As a result, allocating 8 processors on IBM z16 meant that the workload was spread over 2 processor units and in those circumstances we observe the transaction cost can increase. This is discussed in more detail in the “What’s new or changed on IBM z16” section of the “MQ for z/OS on z16” performance report.

Shared queue

Throughput for non-persistent messages in shared queues is dependent on

For messages up to 63KB (64512 bytes):

- z/OS heuristics which can change CF calls from synchronous to asynchronous.
- The type of link(s) between individual z/OS's and the CF(s).
 - This affects the elapsed time to complete CF calls and so influences the heuristics.
 - The CF CPU %BUSY

For messages > 63KB that are stored in DB2:

- As above for up to 63KB messages plus the throughput performance of the DB2 data sharing group tablespace used to store these messages.

The performance effect of these factors can vary significantly from one machine range to another.

For messages > 63KB that are stored in shared message data sets (SMDS):

- Refer to [MQ for z/OS 9.3](#) shared queue - non-persistent server-in-syncpoint workload section in the Regression Appendix.

Maximum persistent message throughput - private queue examples

Using a request/reply workload with no business logic where

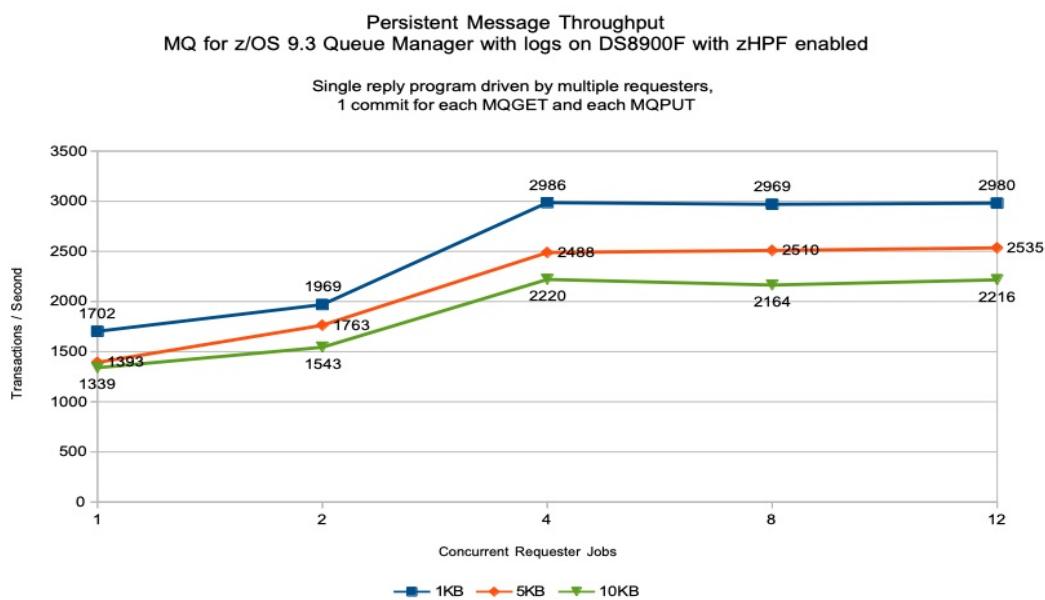
- Requester application(s) use a commit for each MQPUT to a server queue and each MQGET from a reply queue
- Reply application(s) use a commit for each MQGET from the server queue and each MQPUT to the reply queue.
- A request/reply ‘round trip’ uses 2 persistent messages (a request message and a reply message of the same size)
- Thus there is 1 log force for each MQPUT and each MQGET.

We have achieved the following on our 3931-703 z/OS system with DS8900F DASD. On different systems, particularly with different DASD, you may get different results.

As a comparison with older type DASD such as 2105-F20, see “[Maximum Request/Reply throughput \(DS8900F\)](#)”.

Strict ordering - single reply application

The following chart shows how many persistent transactions can be processed by a single reply application with an increasing number of requester applications.



By reviewing the class(3) accounting data for the reply application we can see that it is the limiting factor, as can be seen in data taken from the run using 12 requester tasks with 10KB messages e.g.

== Commit	:	Count	133915,	Avg elapsed	425,	Avg CPU	2
..							
-MQ call-		N	ET	CT	Susp	LOGW	
Put	:	133915	10	9	0	0	
..							
Get	:	133915	7	6	0	0	

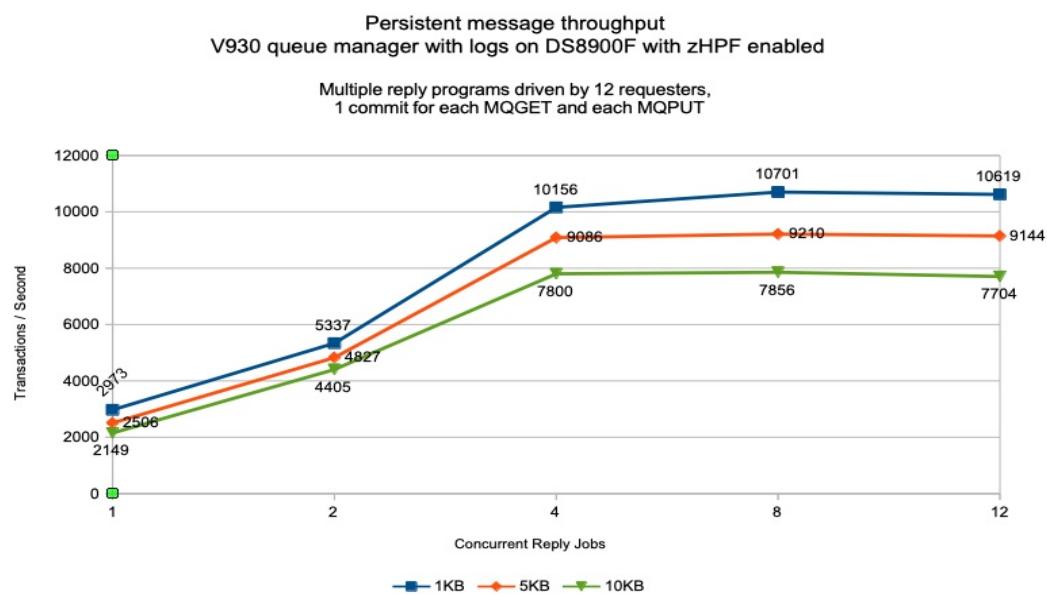
Since the reply transaction is “get, put, commit” we can see that the elapsed time per transaction is: $425 + 10 + 7 = 442$ microseconds.

As the reply application is single threaded, it can process a maximum of 2262 transactions per second (1,000,000 CPU microseconds per second divided by elapsed time of a transaction).

The chart shows 2216 transactions of 10KB were processed with 12 requester tasks running – which shows that the single server is running at nearly 100% of the maximum possible transaction rate - and also shows that the single reply application is spending a significant proportion of its processing time in MQ.

Increasing number of reply applications

The following chart shows persistent message throughput with an increasing number of reply applications and many (12) requester applications.

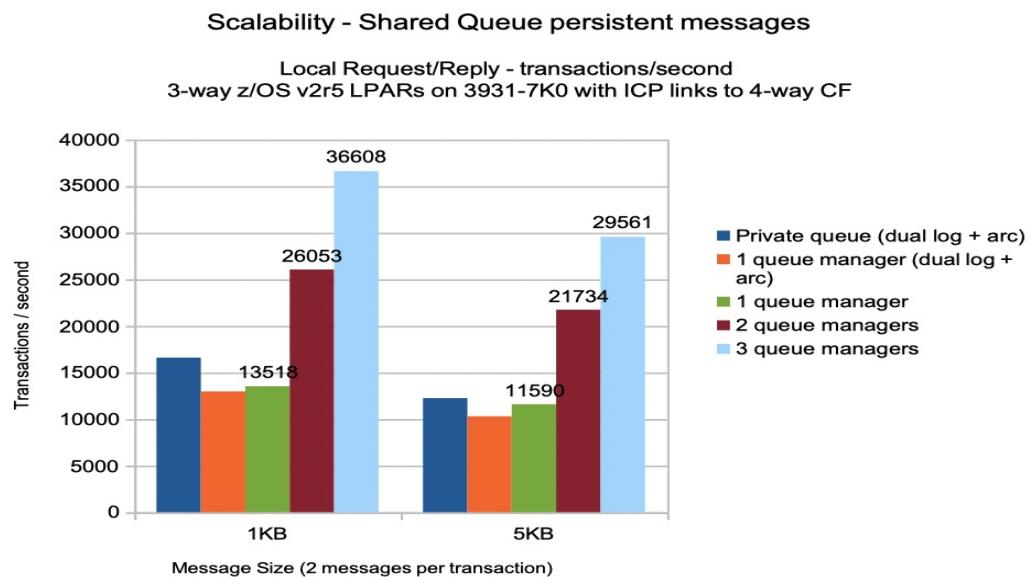


Maximum persistent message throughput - shared queue examples

We have processed more than 84000 1KB persistent messages per second with three queue managers, each on a separate z/OS, using a pair of shared queues when zHPF is enabled.

Persistent message throughput is limited by the rate at which data can be written to the log. Having multiple queue managers each with its own log allows a many times increase in the throughput. Using shared queues means that this many times increase is available through a single queue or set of queues.

The subsequent measurements were run using queue managers running CFLEVEL(5), using default offload thresholds. Due to the size of the messages and the depths of the queues, no offload capacity was required, so the offload medium is irrelevant in these measurements. In all following configurations, zHPF is not enabled.



NOTE: Dual logging and no archiving used unless otherwise stated.

These results were obtained on a parallel sysplex LPARed out of one 3931-7K0 box with a connected DS8900F DASD subsystem. Real production parallel sysplexes would need, and might reasonably be expected to have, sufficient log DASD I/O rate capacity for the required number of logs and archives.

Results on 3931-7K0 systems were obtained using IBM MQ for z/OS 9.3.

“Local Request/Reply” is a set of identical request applications and a set of identical reply applications running on each queue manager such that requesters MQPUT a message to a common server shared queue and MQGET their specific reply message from a common reply shared queue that is indexed by MSGID. The reply applications MQGET the next request message from the common queue, MQPUT the specific reply message to the indexed shared queue and MQCQUIT. Thus there are two messages completely processed (that is created with MQPUT and consumed with MQGET) for each request/reply transaction.

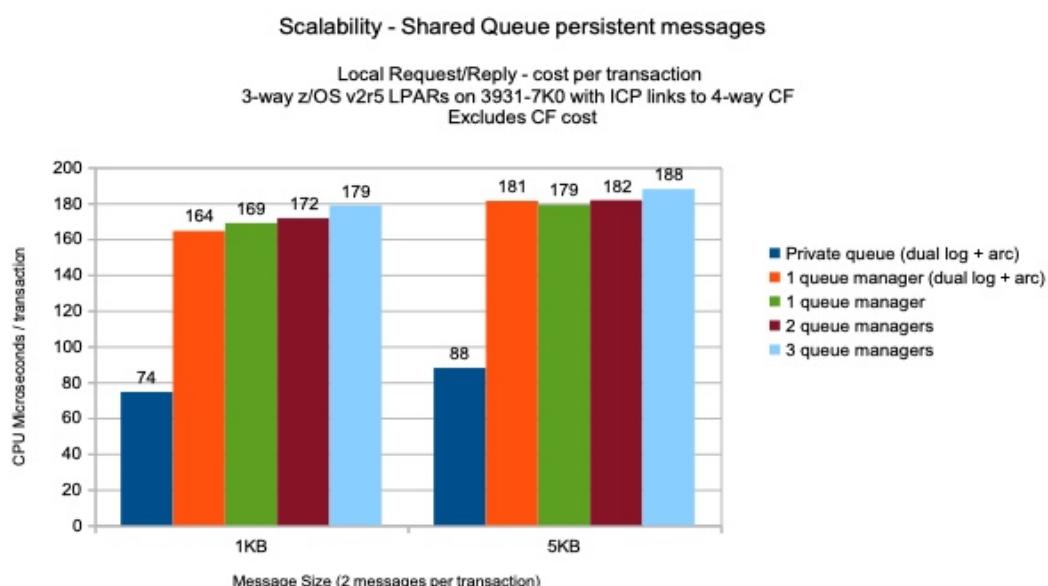
The preceding chart demonstrates the following.

- More than 72000 messages per second (two messages per transaction) with 1KB messages.
- More than 58000 messages per second with 5KB messages.

- Single queue manager workload for shared queue saw the machine running at 72% of capacity compared to 40% of capacity for private queues.
- Archiving versus no archiving can have an effect on throughput of up to 12% for 1 queue manager (dual logging and archiving) versus 1 queue manager.
- Scalability for 1 to 3 queue managers, where there is sufficient:
 - log DASD capacity
 - CF link connectivity
 - CF processing power.

Shared queue persistent message - CPU costs

CPU costs for shared queue are more than usually difficult to estimate from measurements of individual IBM MQ operations. These measurements are the most representative of likely costs for real workloads. This is because they include the interactions between the CF and the queue managers in real life queue sharing under load including waiting MQGET situations. These CPU milliseconds are derived from RMF reported 'MVS BUSY TIME PERC' for each **total system**. Thus they include all system overheads and are larger than would be found from adding 'WORKLOAD ACTIVITY' reported CPU usage by address space.



NOTE: Unless otherwise specified, dual logging and no archiving are used.

Based on the preceding chart, a rule of thumb may be derived that shared queue persistent CPU costs compared to 'best case' private local queue:

- Of the order 225% more than for 1000 byte persistent messages. Each extra queue manager adds 3% to the CPU cost.
- Of the order 200% for 5000 byte persistent messages. Each extra queue manager adds 2% to the CPU cost.

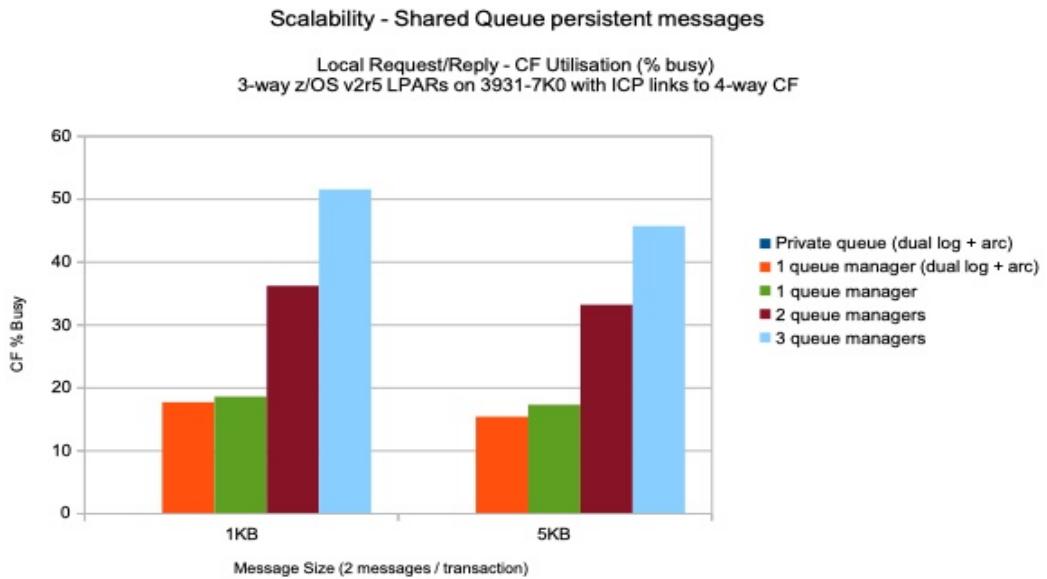
On IBM z16, the percentage increase of shared queue over private queue has increased. This particular private queue configuration cost has decreased by 20% over IBM z15, whereas the shared queue costs have largely remained the same between z15 and z16.

'Best case' means all messages remain within the buffer pool. One of the advantages of shared queue is that there can be enough capacity to keep going even when there is an application instance outage. With private local queues any such outage could severely restrict throughput and/or cause buffer pools to fill with consequently more expensive performance until any backlog is cleared.

Shared queue persistent message - CF usage

MQ calls to the CF are mainly to the application structures. There is no difference in the number and type of calls between persistent and non-persistent messages. There are more calls to the CSQ_ADMIN structure for persistent messages.

The following chart shows the RMF reported 'CF UTILIZATION (% BUSY)' matching the persistent message local request/reply charts above.

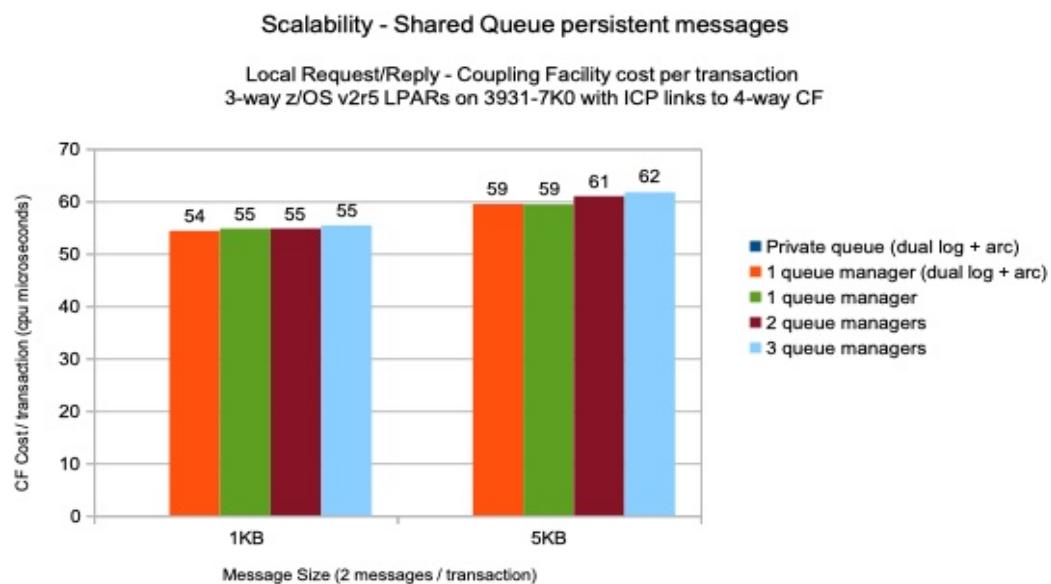


As discussed in "[When do I need to add more engines to my CF?](#)", the CPU "% Busy" figure of the CF remains below 60%, therefore we would surmise that there is sufficient CF capacity for this workload. Were more queue managers to be added driving the workload above 60% we would advise activating an additional engine to the CF. The number of asynchronous calls increases as the CPU usage increases, causing a slower response from the CF.

A rule of thumb for CF cost is about 28 CF microseconds per message (56 CF microseconds per transaction) for 1000 byte messages when using ICP links.

For 5000 byte messages the CF cost is 30 CF microseconds per message when using ICP links. CF costs per message do not change significantly with increasing number of queue managers unless the proportion of asynchronous requests increases as a result of adding additional queue managers.

The following chart shows the cost in the CF per transaction.



Note: Were there insufficient CPU capacity in the CF, the number of asynchronous requests would have increased which would also result in an increase to the CF cost per transaction.

Message ordering - logical groups

Messages on queues can occur, within each priority order, in physical or logical order.

- **Physical order** is the order that in which the messages arrive on a queue.
- **Logical order** is when all of the messages and segments within a group are in their logical sequence, adjacent to each other, in the position determined by the physical position of the first item belonging to the group.

These physical and logical orders can differ because:

- Groups can arrive at a destination at similar times from different applications, therefore losing any distinct physical order
- Even with a single group, messages can get arrive out of order because of re-routing or delay of some of the messages in the group.

There are 2 main reasons for using logical messages in a group:

- You might need to process the messages in a particular order
- You might need to process each message in the group in a related way

Does size of group matter?

When a message group consists of large numbers of messages, there is the potential for deep queues, with many uncommitted messages.

Should the transaction fail, there is the potential for a longer back-out time.

We found that there was no significant overhead associated with putting messages in logical groups.

Large groups of small messages OR small groups of large messages?

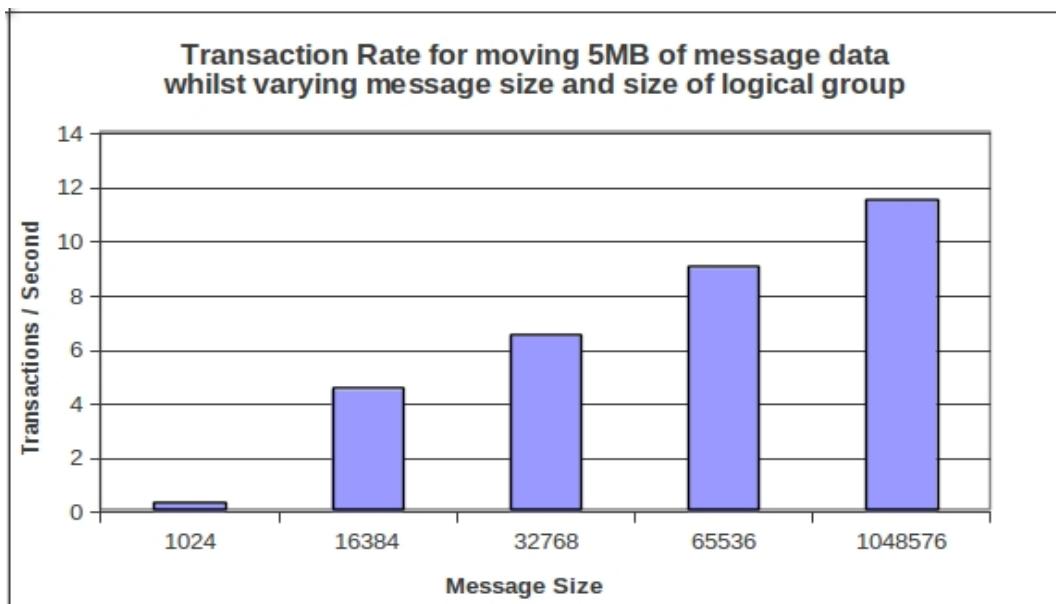
Purely from an MQ perspective it is more cost-efficient and faster to transport small logical groups containing large messages than it is to transport large groups with small messages.

To show a comparison, the following example demonstrates a transaction that moves 5MB of data.

In this example, a transaction is:

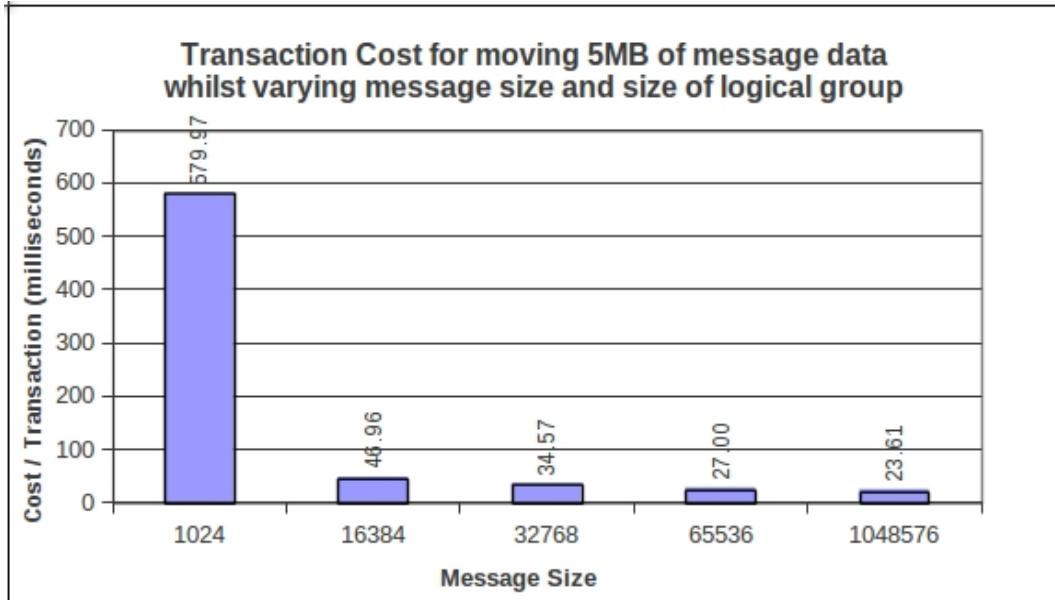
- An application that puts the specified number of persistent messages in a logical group, commits the group and then waits for a single reply message.
- The costs include the application that puts the messages in a logical group and the application that processes the logical group and sends a reply message.
- All measurements were performed on a queue manager using WebSphere MQ v7.0.0 on a single z/OS 1.9 LPAR with 3 dedicated processors from a z10 EC64.

Message Size	Group Size	Message volume transported (message size * group size)
1KB	5000	4.88MB
16KB	320	5MB
32KB	160	5MB
64KB	80	5MB
1MB	5	5MB



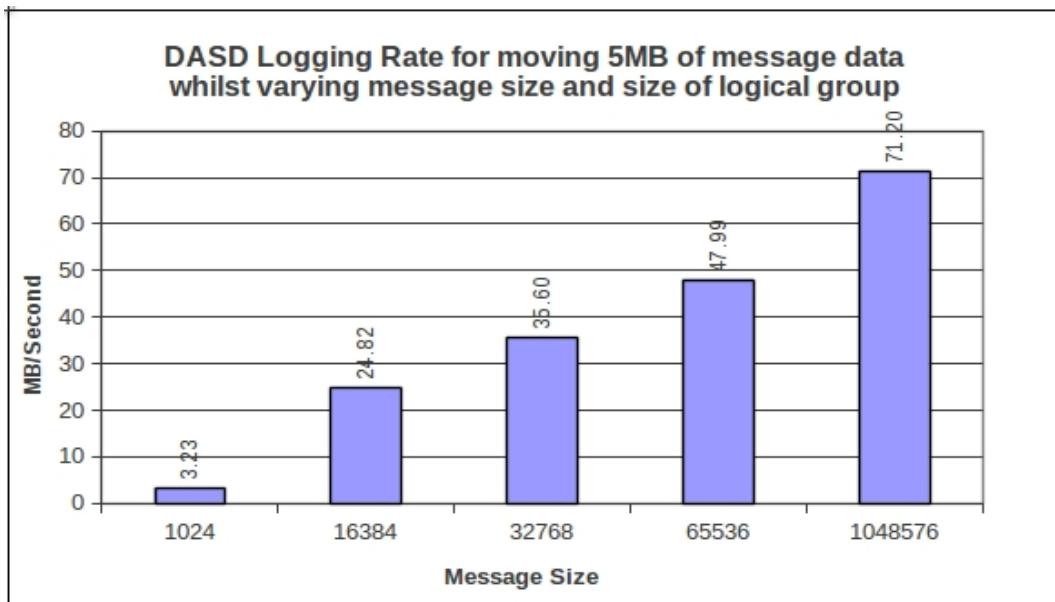
Notes on preceding chart:

- Using a logical group of 5000 with 1KB persistent messages, the achieved transaction rate was 0.3 transactions per second.
- Using 16KB persistent messages with a logical group of 320, the transaction rate increased to 4.5 per second, an increase of 15 times.
- Using 1MB messages with logical groups of 5 messages, the transaction rate increasesd to 11.5 per second, an increase of 38 times.



Notes on preceding chart:

- From an MQ perspective, it is clearly less expensive to process fewer large messages in a logical group than many smaller messages.



Notes on preceding chart:

- When using small (1KB) messages, the rate at which the queue manager can write the logs is much lower (around 16MB/second).
- By contrast, when using large messages, the queue manager can log much faster (70+ MB/second for 1MB messages) and in the above measurements that limit is being reached.
- When batching the application messages into fewer MQ messages, there are less MQ overheads – for example there are only 5 MQMDs logged rather than 5000 MQMDs for a group containing 1KB messages.

Application tuning

How much extra does each *waiting* MQGET cost?

The cost of an MQGET with wait is insignificant - approximately 1 microsecond on a 2817-703. This cost is not dependent on message length, persistence, or whether in or out of syncpoint.

If you have more than one application waiting for a message on a particular queue then every such application will race for any newly arriving message. Only one application will win this race, the other applications will have to wait again. So if you have, for example, five applications all waiting for a message on a particular queue the total cost to get the message is the cost of a successful MQGET, (which does depend on message length, persistence, and whether in or out of syncpoint), plus 5 times 1 CPU microsecond (2817-703).

NOTE: Multiple applications in a get-with-wait can ensure processing throughput even if the cost is slightly higher at low-volume periods.

How much extra does code page conversion cost on an MQGET?

Code page conversion from one single byte character set to another using MQFMT_STRING costs about the same as a basic MQGET out of syncpoint for the same non persistent message size.

DBCS to DBCS conversion costs are of order 4 times a basic MQGET out of syncpoint for the same non-persistent message size.

Event messages

The cost of creating each event message is approximately the same as MQPUT of a small persistent message.

Triggering

For shared queue, there can be a trigger message generated on all queue managers in the QSG. If there are many trigger monitors, only one will get the message, so there may be multiple unsuccessful gets.

Trigger EVERY is suitable when there are low message rates. You should consider having long running transactions to drain the queue.

If trigger every is used with a high message rate, this can add significantly to the CPU cost due to all of the trigger messages generated and any additional processing, for example starting CICS transactions.

What is the cost of creating a trigger or event message?

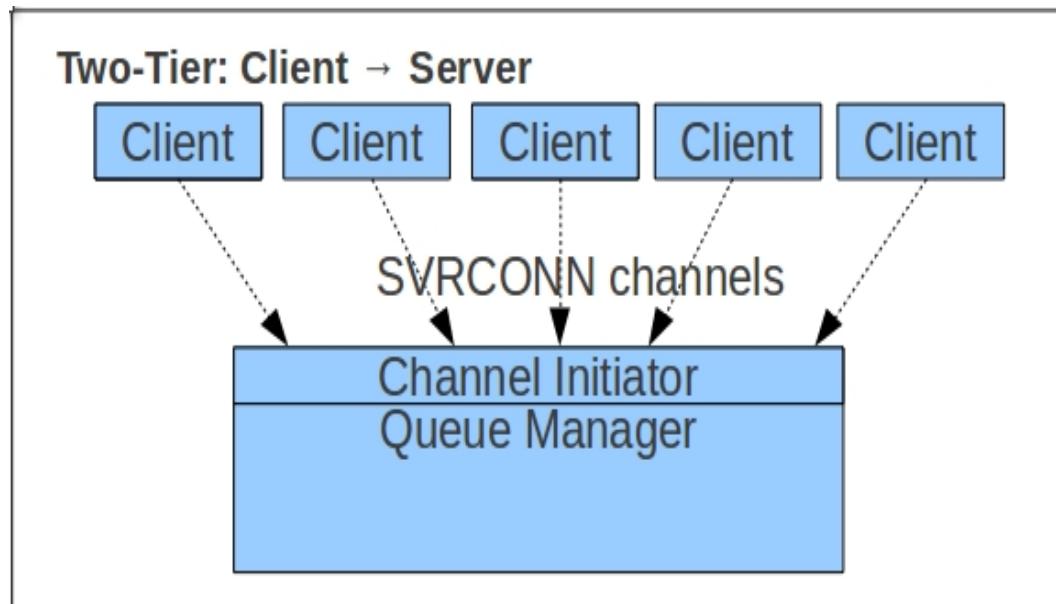
The cost of creating one trigger message is approximately the same as an MQPUT of a small non-persistent message (less than 100 bytes).

When evaluating the cost of triggering, remember to include the cost of the process initiation.

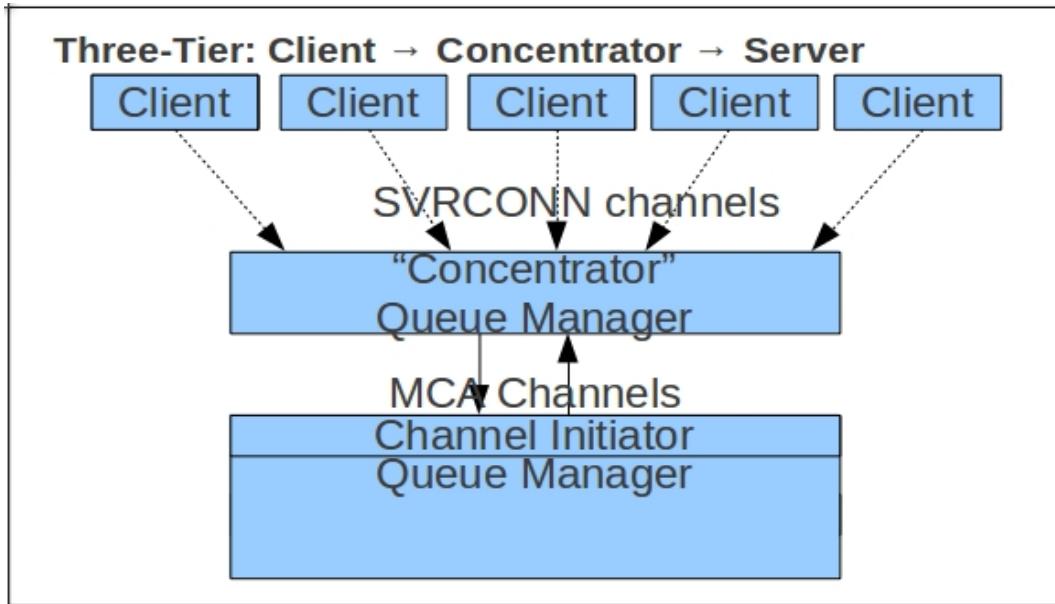
Chapter 8

Two / Three Tier configurations

A typical two-tier configuration is shown in the following diagram. It shows a number of IBM MQ clients connecting using SVRCONN channels directly into the queue manager's channel initiator address space.



A typical three-tier configuration is shown in the following diagram, which uses a concentrator queue manager to accept the SVRCONN connections from the clients and use a single pair of MCA channels to send the messages onto the target queue manager.



Why choose one configuration over the other?

There are several performance factors that should be considered when deciding whether to use a 2 or 3 tier configuration.

- Cost on the z/OS platform
- Achievable rate (latency / persistence / maximum throughput for a channel)
- Number of connecting tasks (footprint)

It should be noted that performance is not the only reason to choose between a 2 or 3 tier configuration. For example if high availability (HA) is a key concern, using a 3 tier model may make the configuration more complex.

Cost on the z/OS platform

The cost of transporting data onto the z/OS platform is often a key criteria when deciding whether to use a two or three tier model.

The cost of connecting directly into a z/OS queue manager via SVRCONN channels can be higher than accessing the z/OS queue manager via MCA channels, from a distributed queue manager, particularly with short-lived connections, where the high connect cost forms a larger proportion of the total cost.

As the number of messages processed during a connection increases, the relative cost of the MQ connect decreases in relation to the total workload. For example:

- If the client connects, puts a 2KB message and then disconnects, the relative cost of the connect is high.
- If the client application were to follow this connect-put-disconnect model but the message is much larger, e.g. a 1MB message, the relative cost of the connect would be much lower.
- Similarly if the client application were to connect, perform many puts and gets, then disconnect, the relative cost of the connect is low.

The key factors relating to cost would be:

- How much work is processed under a single connection, i.e. are the clients long-lived or short-lived?
- The size of the messages.

With very long running transactions or where connection pooling is used resulting in long SVRCONN instances, the costs on z/OS of servicing a SVRCONN connection may be similar to a pair of SDR-RCVR channels.

With regards to MCA channels, as can be seen in the section “[Costs of moving messages to and from z/OS images](#)”, achieving larger batch sizes can reduce transaction cost. This means that more efficient batch sizes can be achieved when multiple clients are putting messages to the distributed queue manager.

Achievable Rate

Volume of data – at channel capacity

The volume of data that each client is likely to send/receive may influence any decision on two or three tier configurations.

A channel, whether a SVRCONN or an MCA channel will have a maximum capacity on any particular network and this will depend on a number of factors including network capacity and latency.

If the combined messaging rate of the clients exceeds the maximum capacity of a single channel, using a 3-tier model may result in reduced throughput, however this can be alleviated by using multiple MCA channels between the distributed and z/OS queue managers.

Latency

Adding the third tier in the form of a distributed queue manager may add latency to the time to transport data from client to z/OS or vice versa. In a low latency network or one with short distance between distributed queue manager and z/OS queue manager this may not be a significant amount.

The transport time between distributed and z/OS queue managers may be further reduced if the distributed queue manager is in a zLinux LPAR and the connection is configured to use HiperSockets.

Persistent Messaging

In the case of persistent messages, adding in a second queue manager will mean that the messages are logged twice, once on the intermediate queue manager and again on the z/OS queue manager, which will add latency to the message flow.

In addition with persistent messages that flow over an MCA channel, each one will be processed serially by the channel initiator adaptor tasks, whereas multiple SVRCONN channels could exploit separate adaptor tasks which in turn can drive the queue managers logger task more efficiently.

Number of connecting tasks

The channel initiator on z/OS has a limit of 9,999 channels, as defined by the MAXCHL attribute, however the number of channels that can be run will depend partially on the size of the messages being used. For further information, please refer to the section “[What is the capacity of my channel initiator task?](#)”

A distributed queue manager is not subject to the same 2GB storage limit as a z/OS channel initiator and as such can be configured to support many times the number of client connections that a z/OS channel initiator can support.

As has been documented in the performance report [MQ for z/OS 9.3](#), the footprint of a SVRCONN channel is 83KB when a 1KB message is being sent and received over a channel configured with SHARECNV(0).

By contrast an MCA channel that sends or receives a 1KB message uses 90KB – however if the connecting task requires a message to be sent and received, it will require 2 channels, with a combined footprint of 180KB.

This suggests that the SVRCONN has a lower footprint, however when multiple clients need to attach, it can be more efficient from a storage usage perspective to use an intermediate queue manager as a “concentrator”, allowing the MCA channels to support multiple clients concurrently.

Measurements

The following configurations were measured:

- Local bindings – requesting application running on z/OS.
- Requesting application uses bindings connection to distributed QM and then use SDR-RCVR channels to interact with z/OS QM.
- Client connection directly to z/OS QM, using SHARECNV(0) and SHARECNV(non-0)
- Client connection to local distributed QM and then use SDR-RCVR channels to interact with z/OS QM.
- Client connection to remote distributed QM and then use SDR-RCVR channels to interact with z/OS QM.

For each of these configurations, two models of requesting application were used:

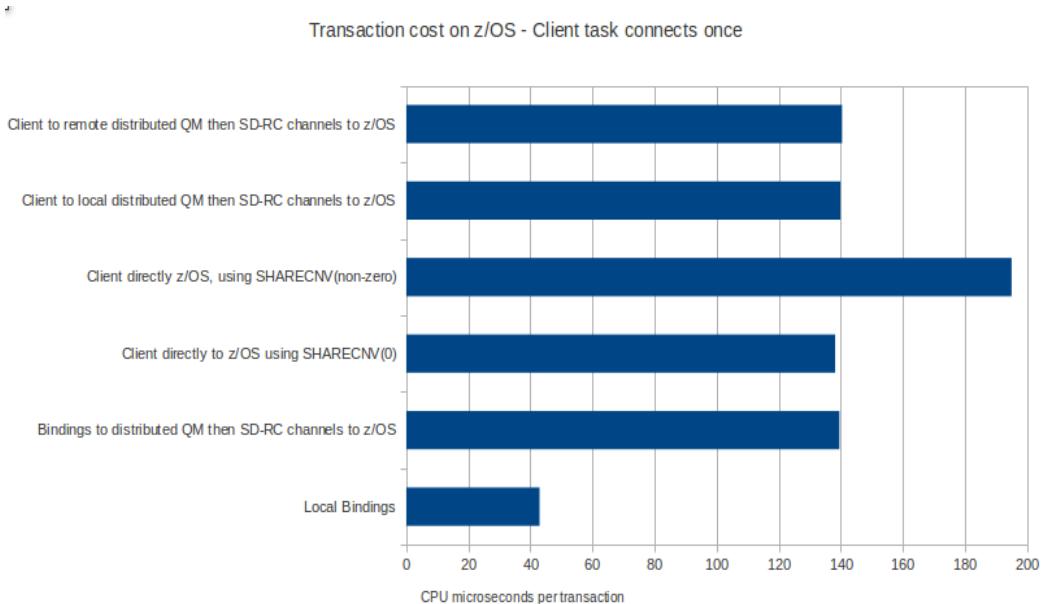
1. Requesting applications use model of: connect, [put, get]*100,000, disconnect.
2. Requesting application use model of: [connect, put, get, disconnect]*100,000.

In each case:

- Messages used were 2KB non-persistent.
- There was a set of long-running batch tasks on z/OS that got the input messages and generated a reply message.

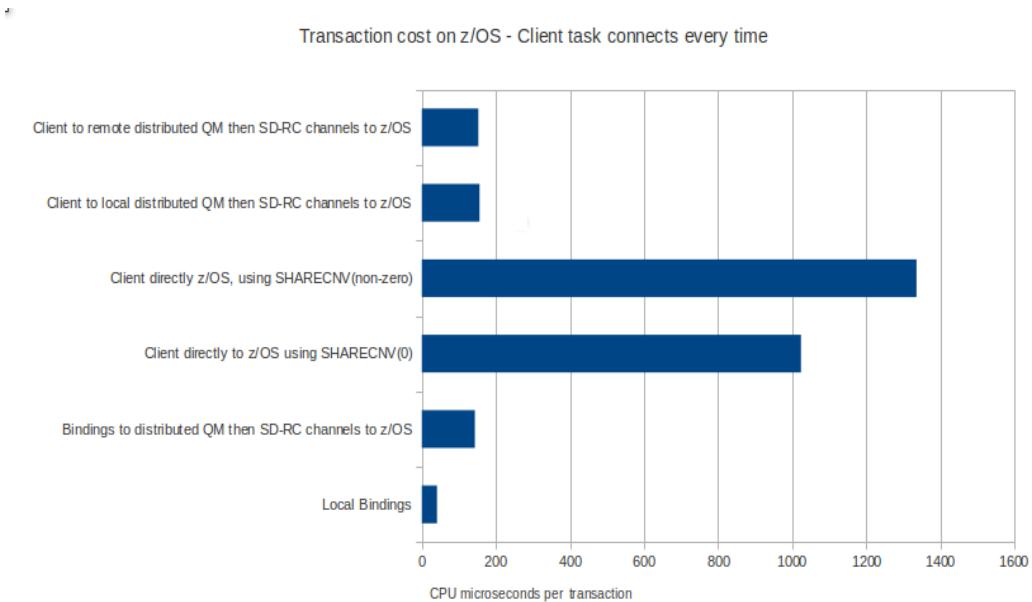
The following 2 charts show the costs of these configurations and include the queue manager, channel initiator, TCPIP and server application costs per message put and got.

Chart: Transaction cost with model: connect, [put, get]*100,000, disconnect



In the preceding chart, the cost of using the SVRCONN channel with SHARECNV(0) in a 2-tier model is similar to that of an application connecting either using bindings or client to a distributed queue manager and that queue manager then using MCA channels into the z/OS queue manager.

Chart: Transaction cost with model: [connect, put, get, disconnect]*100,000



The preceding chart shows that when a client is connecting to the z/OS queue manager for a short period of time, the costs increase significantly.

In each case, connecting a SVRCONN channel with SHARECNV greater than 0 results in additional CPU cost, however this configuration does offer other benefits including asynchronous gets.

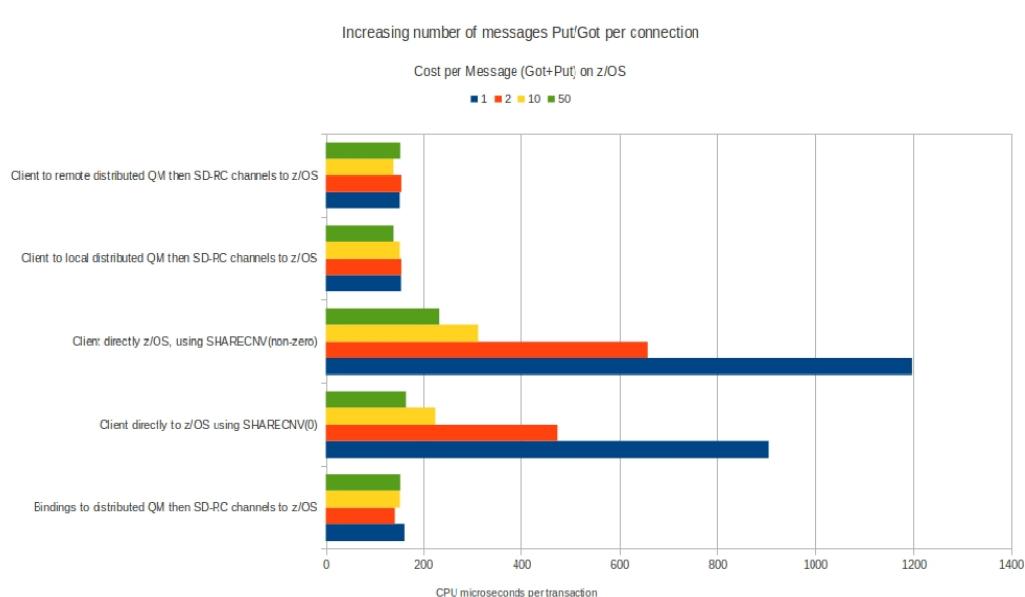
Regarding round-trip times (latency of the workload), the configurations achieved the shortest round-trip time were as per the order they were described earlier i.e. local bindings had shortest round-trip

time and “client to remote distributed queue manager and then use SDR-RCVR channels to z/OS” had the highest round-trip times.

When using shared conversations (SHARECNV larger than 1), the cost of frequently connecting applications can be reduced when there is already 1 or more active conversations when the new connection is requested and there is capacity available (a difference between SHARECNV and the value of CURSHCNV - from a “DISPLAY CHSTATUS” command).

The number of messages processed during the life-span of a channel can affect the cost of a transaction. The following chart provides an example when the number of messages is increased from 1 to 50 messages between connect and disconnect:

Chart: Varying the number of messages processed within connect to disconnect



The preceding chart shows that for a SHARECNV(0) channel, as the number of messages processed between the connect and disconnect increases, the cost per message got and put on z/OS drops significantly, from 905 microseconds when only 1 message is got and put, to 165 microseconds when 50 messages are got and put.

Chapter 9

IMS Bridge: Achieving best throughput

The IBM MQ IMS bridge, known as the IMS Bridge, provides the capability to schedule IMS transactions directly on IMS from an IBM MQ message without the need to use an IMS trigger monitor. The IMS Bridge, which is a component of the IBM MQ for z/OS queue manager, communicates with IMS using the IMS Open Transaction Manager Access (OTMA) service: The IMS bridge is an OTMA client.

When an IMS transaction is driven from a 3270-type screen, any data entered on the screen is made available to the transaction by the IMS GU call. The transaction sends its response back to the terminal using the IMS ISRT call.

An IBM MQ application can cause the same transaction to be scheduled by using the IMS Bridge. An IBM MQ 'request' message destined for IMS, typically with an MQIIH header, is put to an IMS bridge queue. The message is retrieved from the IMS bridge queue by the queue manager and sent to IMS over OTMA logical connections called transaction pipes or tpipes, where the IBM MQ message data becomes input to the IMS GU call. The data returned by the ISRT call will be put into the reply-to queue, where the IBM MQ application can retrieve it using a standard MQGET call. This sequence of events is a typical use of the IMS Bridge and forms the basis of the measurements presented in this section. The IBM MQ request message and its associated reply are referred to as a message-pair or transaction in the rest of this chapter.

This chapter will also provide:

1. Further information on how the IMS Bridge works
2. Suggestions on tuning the IMS subsystem for use with IBM MQ
3. Impact of using Commit Mode 0 or 1
4. How to identify when additional resources are required

Initial configuration

The measurements have been performed using:

- An IMS subsystem
- 3 queue managers in queue sharing group.
- 3 z/OS LPARs running on a 2084-332. Measurements were run on LPAR 1 unless otherwise stated.
 - *LPAR 1 and LPAR 2 are configured with 3 dedicated processors (each LPAR rated as approximately 2084-303)*
 - *LPAR 3 configured with 5 dedicated processors, rated as 2084-305.*
 - *An internal CF with 3 processors.*
- All jobs run were managed using WLM service classes such that the execution velocity is 50% or higher.
- Batch applications were run to drive the workload. Unless otherwise stated, 1KB non-persistent messages were used.
- Unless otherwise stated, the IMS was on the same image as the queue manager and has a varying number of MPRs to process the workload.
- The IMS transaction run issues a “GU” to get the message, then sends the reply using “ISRT” and then if there is another message will issue the “GU” and repeat, otherwise the transaction will end.
- Since the default behaviour of the IMS Bridge is to copy the requester’s message id to the reply messages’ correlation id, it is advisable to define the reply queue with an index type of CORRELID, and perform the get by correlation id.

How does the IMS bridge work?

There are 2 components to the IMS Bridge:

1. Putting messages from IBM MQ into IMS.
2. IMS putting reply messages back to MQ.

Putting messages from IBM MQ into IMS

For each IMS Bridge queue there is a task running in the queue manager. This task gets the message from the MQ queue and puts them into IMS via XCF. The task is effectively issuing an MQGET followed by a put to IMS and then commit.

If the messages are being put to this queue at a high rate, i.e. many applications putting to the queue concurrently, the task may not be able to keep up, resulting in increased queue depths.

When using a shared queue, each additional queue manager often will also have a task per IMS Bridge queue, and will be able to get messages from the queue and send them to IMS.

IMS putting reply messages to IBM MQ

IMS notifies the queue manager that a reply is available and MQ schedules an SRB to process this message. This SRB essentially does an MQOPEN, MQPUT of the reply and an MQCLOSE. If there are multiple replies coming back from IMS, then multiple SRBs can be scheduled and provided there are sufficient processors on the image, these can be run in parallel.

When using shared IMS Bridge queues and multiple queue managers are connected to the IMS Bridge, IMS will usually, but not always, send the reply back to the queue manager that sent the original request.

When an application puts a message to a shared IMS Bridge queue, the queue manager that the application is connected to will not necessarily be the same queue manager that sends the request to IMS. IMS will always put the reply message to the queue manager that put the original message. As a result, this reply message needs to be made available to the original application. By default, IGQ or the mover is used to send the message to the original system.

Using the queue manager option `SQQMNAME(IGNORE)` resolves the shared queue directly to the shared reply queue rather than to any particular queue manager (which would require the message to be moved).

Tuning the IMS subsystem

The IMS subsystem has been configured as described below:

- Checkpoint frequency has been adjusted so that checkpoints are not taken more than once every 10 to 20 minutes. This is controlled using the CPLOG keyword on the IMSCTF system definition macro. Supported values are 1000 to 16,000,000 (default 500,000). To override, modify the DFSPBxxx member of the PROCLIB DD, e.g. “CPLOG=16M”.
- The IMS subsystem has the QBUF attribute set to 256. When the QBUF value was less than the number of MPRs started, there was a significant degradation in throughput.
- Ensure the primary and secondary online logs for the IMS subsystem are large enough to avoid constant log switching and also to avoid being affected by the IMS archive job. Logs have been defined at 1500 cylinders each.
- Only define shared queues to the IMS storage class when they are needed. A fix was applied for PK14315 that wakes up unused IMS Bridge queues every 10 seconds. This is not a large overhead but if there are 100 shared queues defined with the IMS bridge storage class that are not actively being used, there is a small degradation in performance (around 1%).
- Check the IMS attribute QBUF. The IMS trace report provides a field “NUMBER OF WAITS BECAUSE NO BUFFER AVAILABLE”. When this is non-zero, this may be a hint that there is a lack of buffers.
- Check the IMS attribute PSBW. When reviewing some IMS workloads, it was observed than not all of the available MPRs were processing workload. This was because there was insufficient storage set aside to run all of the MPRs concurrently.
- Increase the size of the online datasets. Our default sizes of the DFSOLP* and DFSOLS* datasets meant that they were switching frequently. By increasing them from 50 to 1500 cylinders we reduced the frequency of switching.
- Increase the number of online datasets. When increased throughput occurs, the IMS archive jobs were attempting to process multiple logs in each step. We reached a point where no logs were available. By increasing the number of logs, we were able to prevent waiting for active logs to become available.
- The use of zHPF should be considered in the IMS environment as IMS always logs to disk. In our measurements we saw up to a 20% improvement in throughput.
- A single IMS transaction type defined INQUIRY(YES,NORECOV) for use with non-persistent messages.
- A single IMS transaction type defined INQUIRY(NO,RECOVER) for use with persistent messages.

- Both IMS transaction types are defined with
 - SCHDTYP=PARALLEL
 - PARLIM=0
 - WFI=Y

This will allow the IMS transactions to run concurrently on multiple message processing regions, removing a serialization bottleneck.

Note: Using the command “[/DIS TRAN tranName](#)” will show the value for “PARLM” – and NONE is not the same as 0. NONE means do not allow multiple instances of the transaction.

The message processing regions (MPRs) are started using the IMS performance related options¹ including:

- **DBLDL=<null>** which minimizes the cost of program loading by increasing program directory entries maintained in the MPR. This reduces I/O to program library to obtain the direct address of the program. Default is 20.
- **PRLD=TS** results in PROCLIB DD being searched for member DFSMPLTS. This contains a list of the recommended preloaded MQ modules plus the IMS transactions to be run e.g.

```
CSQACLST, CSQAMLST, CSQAPRH, CSQAVICM, CSQFSALM, CSQQDEFV,
CSQQCONN, CSQQDISC, CSQQTERM, CSQQINIT, CSQQBACK, CSQQCMMT,
CSQQESMT, CSQQPREP, CSQQTTHD, CSQQWAIT, CSQQNORM, CSQQSSOF,
CSQQSSON, CSQQRESV, CSQQSNOP, CSQQCMND, CSQQCVER,
CSQQTMID, CSQQTRGI, CSQBAPPL, IB02INQ, IB02INR
```

- **PWFI =Y** (pseudo wait-for-input) which can potentially reduce CPU time by eliminating the termination and re-scheduling of resources

NOTE: WFI=Y (Wait-For-Input) will keep the program loaded and ready for input. This reduces program start costs but will result in the MPR being dedicated to this transaction. The benefits of WFI=Y are seen when the workload driving the IMS is low and there would be frequent program starts, for example when running with WFI=N and using only 1 batch requester to drive workload using a single shared queue, there were 75 transactions per second. By specifying WFI=Y, the same batch requester was able to drive workload at 880 transactions per second.

When the driving the workload harder, the queue depth for the TPIPE typically is greater than 0 so when the IMS transaction issues a “GU” to get the next message, it is successful. This means that the transaction does not end and have to be restarted for subsequent messages and the MPR does not have to be dedicated to this particular transaction.

¹These options are discussed at [IMS performance options](#)

Use of commit mode

The choice of IMS commit mode specified on the MQIIH structure can affect the performance of the IMS Bridge. Whether to use “Commit-Then-Send” (commit mode 0) or “Send-Then-Commit” (commit mode 1) is dependent on a number of factors, for example, if the application is putting a persistent message (i.e. an “important” message) it may be beneficial to use “Send-Then-Commit” (commit mode 1) – where the IMS message is not committed until the transaction completes. This will ensure that in the event of a failure, the transaction will either have completed successfully or will not have committed the work.

The flows involved for these commit modes are shown below:

Commit Mode 0 (Commit-Then-Send)

1. **IMS Bridge** puts message to TPIPE
2. **IMS** acknowledges
3. **MPR** runs application that issues GU + ISRT, then issues syncpoint
4. **IMS** output is enqueued to TPIPE
5. **MPR** transaction completes
6. **IMS** output sent (with response requested)
7. **MQ** acknowledges
8. **IMS** output de-queued

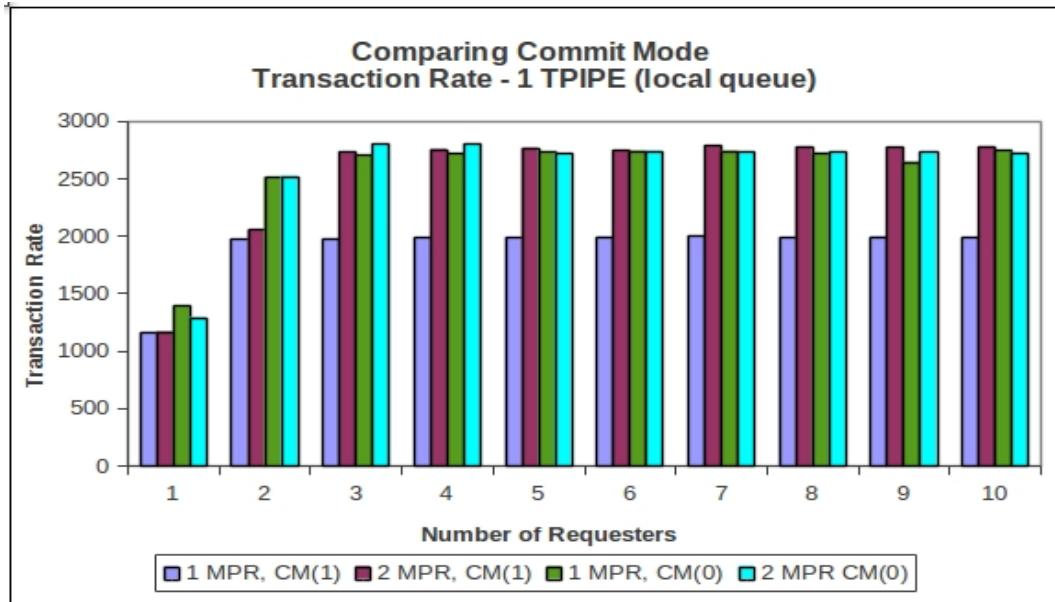
Commit Mode 1 (Send-Then-Commit)

1. **IMS Bridge** puts message to TPIPE
2. **MPR** runs application that issues GU + ISRT
3. **MPR** starts syncpoint
4. **IMS** output is sent to TPIPE, no response requested
5. **IMS** confirms commit, and completes the syncpoint
6. **MPR** transaction completes

For commit mode 0 (CM0), the message processing region (MPR) is able to process the next message before IMS sends the response to the MQ queue manager.

For commit mode 1 (CM1) case, the MPR is only available to process the next message when the transaction completes – i.e. step 6, which is after IMS has sent the output. This means the MPR is involved in each transaction for longer and this may mean additional MPRs are required sooner.

The following chart compares the transaction rate when increasing the number of batch requesters putting 1K non-persistent messages to a single local queue that has been defined with the IMS Bridge storage class. The commit mode is varied to show how CM1 can restrict the transaction rate when there are insufficient message processing regions.



Notes on chart:

- In none of the above measurements is the machine running at 100% capacity.
- When there is only 1 MPR servicing the workload, a peak transaction rate of 2000/second is seen for the applications using CM1. By contrast, when running with 1 MPR and CM0, the peak transaction rate is 2750/second.
- By adding a second MPR for the CM1 workload, the peak transaction rate is able to track the CM0 workload.
- The chart also shows that in this environment², adding a second MPR for the CM0 workload gave little benefit.

² These measurements were run on a 2084-303 and the machine was running at approximately 90% of capacity.

Understanding limitations of the IMS bridge

There are a number of components involved when using the IMS bridge that need to be considered when attempting to achieve the best transaction rate for an application.

Consider the scenario where the achieved transaction rate does not match the required rate. All of the following components could be affecting the transaction rate:

- **IMS**

- Are the message processing regions running at capacity? There may not be sufficient message processing regions available.
- Can the transaction process multiple requests? When the transaction finishes without checking for subsequent requests, there can be a significant effect on the transaction rate if the next transaction has to be loaded.

- **IBM MQ**

- Is there a build up of messages on the IMS Bridge queues? This may be for a number of reasons including:
- Either the IMS subsystem or message processing regions are not being given sufficient priority.
- The MQ task is unable to get the messages fast enough. Using more IMS Bridge queues may help – alternatively if using shared queues, additional queue managers in the queue sharing group may help.
- Is there a build-up of messages on the reply-to queue? Is the reply-to queue indexed appropriately, e.g. if getting using the CORRELID, verify that the queue is indexed by CORRELID.

- **CPU**

- Is the machine running at peak capacity? A quick indication can be seen from the “Active Users” panel in SDSF – e.g. via “S.DA”. For a longer term view, the “LPAR Cluster Report” report from RMF can be used to determine whether the machine is running at or close to 100% capacity.

L P A R C L U S T E R R E P O R T ^a								
z/OS V1R9		SYSTEM ID MV25						
----- WEIGHTING STATISTICS -----								RPT VERSION V1R9 RMF TIME 22.12.13
----- PROCESSOR STATISTICS -----								
CLUSTER	PARTITION	SYSTEM	INIT	MIN	MAX	DEFINED	ACTUAL	LBUSY PBUSY
PLEX3	MVS2A	MV2A	DED			3	3	99.97 9.67
	MVS2B	MV2B	DED			5	3	97.52 9.44
	MVS25	MV25	DED			3	3	100.0 9.68

^a Some detail has been removed from the LPAR cluster report shown since the system is using dedicated processors.

- **Coupling Facility**

- Is the coupling facility running at an optimum level – use of the RMF III reports will show how utilised the coupling facility CPUs are.

- What type of links between the z/OS image and the coupling facility are in place – are they ICP links or are they a physical cable?
- Are the coupling facility structures duplexed and how far apart are they located?
- For more information on coupling facility considerations, refer to the “[Coupling Facility](#)” chapter.

- **XCF**

- Since the IMS Bridge function in the queue manager uses XCF to pass requests to the IMS, there will be a requirement to ensure the XCF address space is running optimally. The XCF address space allocates the Couple Datasets (XCF, Logger, CFRM, WLM, ARM and OMVS). The document “[Parallel Sysplex Performance: XCF Performance Considerations](#)”, gives guidelines on tuning XCF which may be of benefit to improving the IMS Bridge throughput.

The following sections aim to help to identify where the constraints may lie by giving guidance on how to determine:

- When the constraint is due to the message processing region.
- When the constraint is due to the IBM MQ task that puts the message onto the IMS Bridge queue.

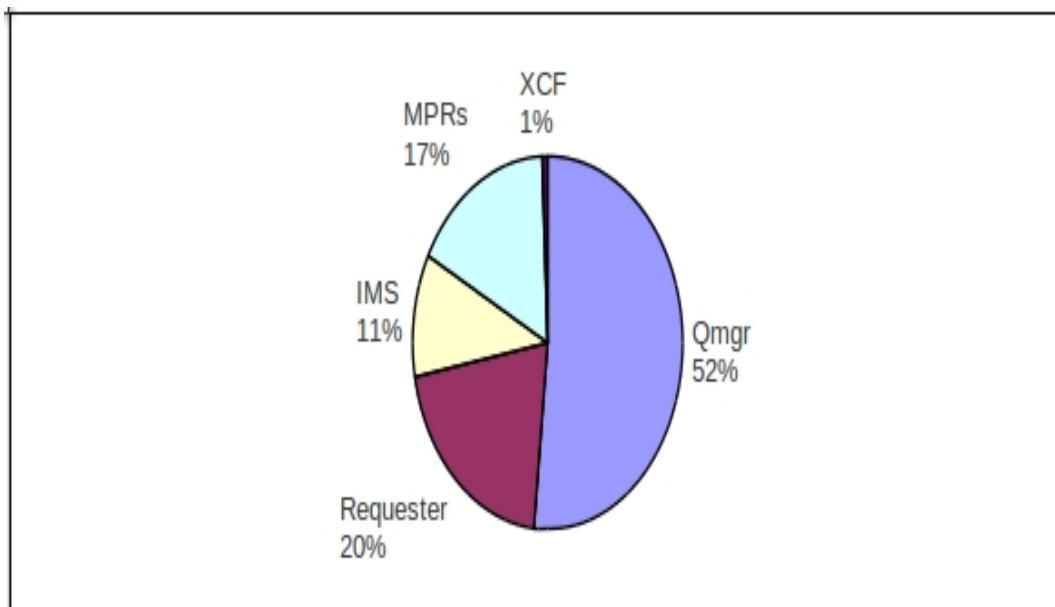
When do I need more message processing regions?

By providing a message processing region (MPR) that has the IMS transaction to be run pre-loaded and in a wait-for-input (WFI), there is a significant reduction in the time taken to schedule the IMS transaction.

The following trace examples are taken when the following environment is used:

- A single 2084-303 image
- A single queue manager in a queue sharing group
- A single IMS control region
- 2 MPRs configured as detailed at the top of this section, i.e. the transactions being run are set to be WFI=Y.
- 6 batch applications each putting to a single IMS Bridge shared queue and getting a reply message using the CORRELID from a corresponding reply-to shared queue that is indexed by CORRELID.
- Commit Mode 1 – “Send then Commit” was specified in the MQIIH data structure.
- The machine was seen to be running at 92% capacity and the overall transaction rate was 1803 transactions per second.

The following shows where the work was done in achieving the throughput mentioned above.



To identify whether the workload was constrained by lack of MPRs, the IMS trace function was enabled using “**/TRACE SET ON MONITOR ALL INTERVAL 60**” and the IMS program DFSUTR20 was used to print the records e.g.

```
//JOBLIB DD DISP=SHR,DSN=IMS.VXX.DBDC.SDFSRESL  
// EXEC PGM=DFSUTR20,REGION=512K  
//SYSPRINT DD SYSOUT=*  
//SYSUT1 DD DISP=SHR,DSN=IMSDATA.IMXX.IMS MON  
//ANALYSIS DD *  
DLI  
/*
```

This produced a number of useful reports³ including:

1. Run Profile
2. Call Summary
3. Region Summary Report
4. Region IWAIT (IMS Wait)
5. Program Summary
6. Program I/O

Understanding the trace reports - run profile

IMS MONITOR *** RUN PROFILE ***
TRACE ELAPSED TIME IN SECONDS.....59.9
TOTAL NUMBER OF MESSAGES DEQUEUED.....107692
TOTAL NUMBER OF SCHEDULES.....2
NUMBER OF TRANSACTIONS PER SECOND.....1794.8
TOTAL NUMBER OF DL/I CALLS ISSUED.....215380
NUMBER OF DL/I CALLS PER TRANSACTION.....1.9
NUMBER OF OSAM BUFFER POOL I/O'S.....0, 0.0 PER TRANSACTION
NUMBER OF MESSAGE QUEUE POOL I/O'S.....115, 0.0 PER TRANSACTION
NUMBER OF FORMAT BUFFER POOL I/O'S.....0, 0.0 PER TRANSACTION

Notes on ‘run profile’ report

- The number of “messages dequeued” is the number of scheduled PSBs.
- The number of schedules (2) indicates that only 2 transactions were started during the monitoring period – this is as expected since the transactions are defined with WFI=Y and there were 2 MPRs started
- The “transactions per second” is the number of PSB schedules per second, so in the above example there are 1794.8 PSBs being scheduled each second for the lifetime of the monitoring period (59.9 seconds). This is close to the previously reported transaction rate of 1803 transactions/second – which was calculated using figures output from the application programs.

³ Information on these reports can be found at
[IMS Monitor Reports](#)

Understanding the trace reports – call summary

IMS MONITOR *** CALL SUMMARY ***						. . . ELAPSED TIME NOT IWAIT TIME . . .					
PSB	NAME	PCB	NAME	FUNC	NO. SEGMENT	STAT	CODE	CALLS	IWAITS	IWAITS/ CALL	MEAN	MAXIMUM	MEAN	MAXIMUM
IB02INQ	I/O	PCB	ISRT	(..)			107691	0	0.00	11	2640	11	2640
			GU	(..)			107689	10	0.00	687	28874	686	28874
I/O PCB SUBTOTAL								215380	10	0.00		349		349
PSB TOTAL								215380	10	0.00		349		349

Notes on ‘call summary’ report

- The “Call Summary” report shows how simple the IMS transaction is – it issues a “GU” call to get the MQ message and issues a “ISRT” to send the reply message.
 - The number of IWAITS (IMS waits) for the GU function is 10, which since there were 107,691 calls indicates that the MPR was being driven hard enough such that it was not waiting for work.

Understanding the trace reports – region summary report

SCHEDULING AND TERMINATION			ELAPSED TIME.....			NOT IWAIT TIME(ELAPSED-IWAIT)		
OCCURRENCES	TOTAL	MEAN	MAXIMM	TOTAL	MEAN	MAXIMUM		
SCHEDULE TO FIRST CALL								
**REGION	1	1	3235	3235	3235	3235		
**REGION	4	1	1007	1007	1007	1007		
**TOTALS		2	4242	2121				
ELAPSED EXECUTION								
**REGION	1	1	38219948	38219948	38219948	38219948		
**REGION	4	1	38117765	38117765	38117765	38117765		
**TOTALS		2	76337713	38168856				
DL/I CALLS								
**REGION	1	107821	37679783	349	27264	37668306	349	
**REGION	4	107559	37571817	349	28874	37568048	349	
**TOTALS		215380	75251600	349		75236354	349	
IDLE FOR INTENT								
CHECKPOINT	6	143571	23928	29396	143571	23928	29396	
REGION OCCUPANCY								
**REGION	1	63.7%						
**REGION	4	63.5%						

Notes on ‘call summary’ report

- The “Checkpoint” section indicates there were 6 checkpoints taken during the monitoring period (as seen below). Ideally the checkpoint frequency will be between 10 and 20 minutes, so in the above example, we are taking checkpoints too frequently. Changing the IMS “CPLOG” value should cause the checkpoint frequency to change.
- The “Region Occupancy” section indicates the ratio of the elapsed time when the thread is active to the trace interval. This suggests that the MPRs are running at 63% of capacity and therefore it should be possible to drive this workload harder than in this example. It also suggests that adding MPRs would not necessarily improve the throughput for this scenario.

IMS Control Region issuing checkpoints whilst monitoring running

12.24.54	STC00997	DFS2212I DC MONITOR STARTED	IM9A
12.25.03	STC00997	DFS2716I NO MSDBS FOUND - NO MSDB CHECKPOINT TAKEN IM9A	
12.25.03	STC00997	DFS994I *CHKPT 08296/122503**SIMPLE*	IM9A
12.25.03	STC00997	DFS3499I ACTIVE DDNAMES: MODBLKSA IMSACBA FORMATA MODSTAT ID: 0 IM9A	
12.25.03	STC00997	DFS3804I LATEST RESTART CHKPT: 08296/122436, LATEST BUILDQ CHKPRT: 08296/120523 IM9A	
12.25.13	STC00997	DFS2716I NO MSDBS FOUND - NO MSDB CHECKPOINT TAKEN IM9A	
12.25.13	STC00997	DFS994I *CHKPT 08296/122513**SIMPLE*	IM9A
12.25.13	STC00997	DFS3499I ACTIVE DDNAMES: MODBLKSA IMSACBA FORMATA MODSTAT ID: 0 IM9A	
12.25.13	STC00997	DFS3804I LATEST RESTART CHKPT: 08296/122503, LATEST BUILDQ CHKPRT: 08296/120523 IM9A	
12.25.22	STC00997	DFS2716I NO MSDBS FOUND - NO MSDB CHECKPOINT TAKEN IM9A	
12.25.22	STC00997	DFS994I *CHKPT 08296/122522**SIMPLE*	IM9A
12.25.22	STC00997	DFS3499I ACTIVE DDNAMES: MODBLKSA IMSACBA FORMATA MODSTAT ID: 0 IM9A	
12.25.22	STC00997	DFS3804I LATEST RESTART CHKPT: 08296/122513, LATEST BUILDQ CHKPRT: 08296/120523 IM9A	
12.25.31	STC00997	DFS2716I NO MSDBS FOUND - NO MSDB CHECKPOINT TAKEN IM9A	
12.25.31	STC00997	DFS994I *CHKPT 08296/122531**SIMPLE*	IM9A
12.25.31	STC00997	DFS3499I ACTIVE DDNAMES: MODBLKSA IMSACBA FORMATA MODSTAT ID: 0 IM9A	
12.25.31	STC00997	DFS3804I LATEST RESTART CHKPT: 08296/122522, LATEST BUILDQ CHKPRT: 08296/120523 IM9A	
12.25.41	STC00997	DFS2716I NO MSDBS FOUND - NO MSDB CHECKPOINT TAKEN IM9A	
12.25.41	STC00997	DFS994I *CHKPT 08296/122541**SIMPLE*	IM9A
12.25.41	STC00997	DFS3499I ACTIVE DDNAMES: MODBLKSA IMSACBA FORMATA MODSTAT ID: 0 IM9A	
12.25.41	STC00997	DFS3804I LATEST RESTART CHKPT: 08296/122531, LATEST BUILDQ CHKPRT: 08296/120523 IM9A	
12.25.50	STC00997	DFS2716I NO MSDBS FOUND - NO MSDB CHECKPOINT TAKEN IM9A	
12.25.50	STC00997	DFS994I *CHKPT 08296/122550**SIMPLE*	IM9A
12.25.50	STC00997	DFS3499I ACTIVE DDNAMES: MODBLKSA IMSACBA FORMATA MODSTAT ID: 0 IM9A	
12.25.50	STC00997	DFS3804I LATEST RESTART CHKPT: 08296/122541, LATEST BUILDQ CHKPRT: 08296/120523 IM9A	
12.25.54	STC00997	DFS2213I DC MONITOR STOPPED - (TIME INTERVAL ELAPSED) IM9A	

Notes: This extract confirms that there were 6 checkpoints taken during the period of monitoring as seen in the previous “Region Summary Report”.

Understanding the Trace reports – Region IWAIT Report

There will be a “Region IWAIT” report for each region, so in this example there will be 2 reports.

IMS MONITOR *** REGION IWAIT ***					
		TOTAL	MEAN	MAXIMUM	FUNCTION MODULE
**REGION	1 OCCURRENCES				
SCHEDULING + TERMINATION					
...SUB-TOTAL...					
...TOTAL...					
DL/I CALLS	4	11477	2869	9734	DD=LGMSG
...TOTAL...	4	11477	2869		QMCG
CHECKPOINT					
...TOTAL...					

IMS MONITOR *** REGION IWAIT ***					
		TOTAL	MEAN	MAXIMUM	FUNCTION MODULE
**REGION	4 OCCURRENCES				
SCHEDULING + TERMINATION					
...SUB-TOTAL...					
...TOTAL...					
DL/I CALLS	6	3769	628	995	DD=LGMSG
...TOTAL...	6	3769	628		QMCG
CHECKPOINT					
...TOTAL...					

Notes: These reports suggest that region 1 is waiting on average 4.5 times longer than region 4 on the few occasions that the regions are in a wait (a total of 10 times). It also suggests that all of the waits were for the LGMSG dataset.

Understanding the trace reports – Program Summary Report

IMS MONITOR *** PROGRAM SUMMARY ***									
		NO.	TRANS.	CALLS	I/O	TRAN.	CPU	ELAPSED	SCHED TO
PSBNAME	SCHEDS.	DEQ.	/TRAN	IWAITS	DEQD.	TIME	.	TIME	1ST CALL
IB02INQ	2	107692	215380	1.9	10	0.0	3846.0	26	38168856
**TOTALS	2	107692	215380	1.9	10	0.0	3846.0	26	38168856

Notes: This reports confirm that there were 10 waits for I/O (for DD=LGMSG). It also tells us that in the monitoring period, 107,692 transactions were run over the 2 message processing regions.

Understanding the trace reports – Program I/O Report

IMS MONITOR *** PROGRAM I/O ***						
		IWAITS	TOTAL	MEAN	MAXIMUM	DDN/FUNC MODULE
PSBNAME	PCB NAME					
IB02INQ	I/O PCB	50600	21753542	429	28525	**W F I
		50379	21858044	433	23535	**W F I
		10	15246	1524	9734	LMSG QMG
PCB TOTAL		10	15246	1524		
PSB TOTAL		10	15246	1524		
GRAND TOTAL		10	15246	1524		

Notes: This report suggests that the IMS transaction “IB02INQ” has had a significant number of IMS waits (100,979) with an average time of 429-433 microseconds. This again suggests that there are sufficient MPRs to support the MQ workload, or conversely that there was insufficient workload to drive the MPR at capacity.

As an example of when additional MPRs may be required, the following 2 reports were taken when running with 16 batch applications using the same TPIPE.

16 Batch Applications, 1 PIPE, 4 MPRs

IMS MONITOR *** PROGRAM I/O ***						
PSBNAME	PCB NAME	IWAITS	TOTAL	MEAN	MAXIMUM	DDN/FUNC MODULE
IB02INQ	I/O PCB	41	24830	605	1628	**W F I
		41	33346	813	3314	**W F I
		43	25763	599	2063	**W F I
		27	12985	480	1931	**W F I
		8	15786	1973	10010	LGMMSG QMG
PCB TOTAL		8	15786	1973		
PSB TOTAL		8	15786	1973		
GRAND TOTAL		8	15786	1973		

16 Batch Applications, 1 PIPE, 5 MPRs

IMS MONITOR *** PROGRAM I/O ***						
PSBNAME	PCB NAME	IWAITS	TOTAL	MEAN	MAXIMUM	DDN/FUNC MODULE
IB02INQ	I/O PCB	1593	900430	565	5323	**W F I
		1530	853962	558	3391	**W F I
		1525	886910	581	8103	**W F I
		1595	887466	556	5685	**W F I
		1613	899249	557	4565	**W F I
		10	6843	684	1124	LGMMSG QMG
PCB TOTAL		10	6843	684		
PSB TOTAL		10	6843	684		
GRAND TOTAL		10	6843	684		

Notes: In both cases, the number of IMS waits is significantly less than in the original example. The numbers in blue correspond to the number of message processing regions active.

When there are 4 MPRs servicing the MQ workload, a very low number of IWaits can be seen. This would indicate that the MPRs are nearly always busy, and adding additional MPRs may be of benefit. There is also a small number of waits (8) on the LGMMSG dataset.

When there are 5 MPRs servicing the workload, the number of IWaits increases which suggests there is spare capacity.

To confirm this, a look at the “Region Occupancy” section of the “Region Summary” report, shows that for 4 MPRs, each region was running at 99.7% whereas when running with 5 MPRs, the regions ran at 98.3% - again showing there is some, but not much, spare capacity.

When do I need more TPIPEs?

At some point, the task running in the queue manager that is getting messages from the request queue and putting to IMS cannot run any faster. At this point, additional throughput may be achieved by using multiple IMS Bridge queues (which will each have a separate queue manager task).

Alternatively, if shared queues are in use, the addition of another queue manager in the QSG will provide another task to get the message from the queue and pass to IMS.

To determine whether more TPIPEs will benefit the throughput, the IMS region will again need to be monitored to determine whether the MPRs are working efficiently. In addition, the MQ queues need to be checked to see if a backlog is building either on the request or the reply queue.

This example is based on 2 sets of measurements:

Both measurements used local IMS Bridge queues with the workload running on LPAR 1 (rated as a 2084-303).

Commit Mode	0 Commit-Then-Send	1 Send-Then-Commit
MPR	2	2
TPIPEs	1	1
Requester Tasks	6	6
Persistent Messages	No	No
Transaction Rate	2700 / sec	2747 / sec
CPU Utilisation	82.5%	87%
Cost / Transaction	917 microseconds	950 microseconds

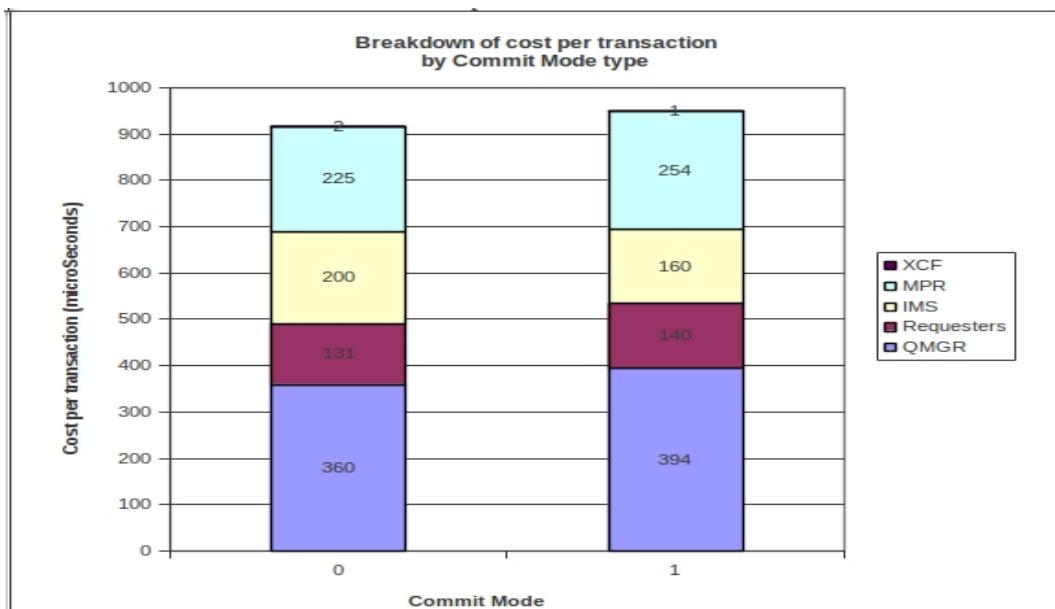
What information can we get from the above table?

- The workload is not CPU-constrained.
- Whilst CM1 is achieving a higher transaction rate, the cost per transaction is also higher than for CM0.

The following chart shows where the costs are incurred in these 2 measurements.

Compared to commit mode 0 “Commit-Then-Send”, the commit mode 1 “Send-Then-Commit” measurements show:

- 10% increase in the cost of the queue manager processing the transaction over commit mode 0 “Commit-Then-Send”
- 13% increase in the cost of the MPR processing the transaction.
- 20% decrease in the cost to the IMS control region
- A 3.5% overall increase in cost per transaction.



Looking at the trace reports for these 2 measurements shows the Region Occupancy for the MPRs at:

- 31.2% and 29.9% for the commit mode 0 “Commit-Then-Send” measurement.
- 72.3% and 72.3% for the commit mode 1 “Send-Then-Commit” measurement.

Looking at the “Program I/O” report for the Commit Mode 0 measurement:

IMS MONITOR *** PROGRAM I/O ***						
.....IWAIT TIME.....						
PSBNAME	PCB NAME	IWAITS	TOTAL	MEAN	MAXIMUM	DDN/FUNC ...
IB02INQ	I/O PCB	82718	41191904	497	192534	**W F I
		79081	42015547	531	168595	**W F I
		52	111728	2148	25744	LGMSG

This above extract shows that there are a significant number of IMS waits for each MPR, average 0.5 milliseconds.

Similarly the “Program I/O” report for Commit Mode 1 shows:

IMS MONITOR *** PROGRAM I/O ***						
.....IWAIT TIME.....						
PSBNAME	PCB NAME	IWAITS	TOTAL	MEAN	MAXIMUM	DDN/FUNC ...
IB02INQ	I/O PCB	79882	16582118	207	29985	**W F I
		79855	16592048	207	29200	**W F I
		10	17574	1757	13233	LGMSG

This extract shows that there are a significant number of IMS waits for each MPR, average 0.2 milliseconds.

This suggests that the MPRs have the capacity to be driven at higher transaction rate.

Since we also know that there is more capacity available on the LPAR, something else is causing the constraint.

By issuing the queue manager command “RESET QSTATS” against the request and the reply queues, firstly to reset and then secondly (a period of time later) to review, it was seen that the request queue has a HIQDEPTH of 6 for both commit modes. The reply queue had a HIQDEPTH of 1 for commit mode 0 and 4 for commit mode 1.

That the request queue had a HIQDEPTH of 6 suggests that the queue manager task that gets the message from the request queue and put to the IMS Bridge is not able to keep pace with the workload.

In the case of the reply queue, the values for HIQDEPTH show that for commit mode 0, the replies are being gotten by the batch applications as quickly as the messages are put.

The tests were re-run using 2 TPIPEs, to see if allowing the queue manager a second task to get messages from the request queues will help drive the workload at a higher rate.

Commit Mode	0 Commit-Then-Send	1 Send-Then-Commit
MPR	2	2
TPIPEs	2	2
Requester Tasks	6	6
Persistent Messages	No	No
Transaction Rate	2852 / sec	3017 / sec
CPU Utilisation	89.5%	92.57%
Cost / Transaction	941 microseconds	920 microseconds

Looking at the trace reports for these 2 measurements shows the Region Occupancy for the MPRs at:

- 33.6% and 34% for the Commit Mode 0 “Commit-Then-Send” measurement.
- 99.7% and 99.7% for the Commit Mode 1 “Send-Then-Commit” measurement.

Looking at the “Program I/O” report for the Commit Mode 0 (Commit-then-Send) measurement:

IMS MONITOR *** PROGRAM I/O ***						
.....IWAIT TIME.....						
PSBNAME	PCB NAME	IWAITS	TOTAL	MEAN	MAXIMUM	DDN/FUNC ...
IB02INQ	I/O PCB	85064	39500980	464	13979	**W F I
		84605	39736498	469	86560	**W F I
		82	167153	2038	37939	LGMSG

This shows a slight increase in the number of IMS waits (from 161,799 to 169,669) but the average time has also decreased slightly (from 500 microseconds to 467 microseconds).

For Commit Mode 1 (Send-then-Commit), there is a marked improvement –which can be seen in the “Program I/O” report following:

IMS MONITOR *** PROGRAM I/O ***						
.....IWAIT TIME.....						
PSBNAME	PCB NAME	IWAITS	TOTAL	MEAN	MAXIMUM	DDN/FUNC ...
IB02INQ	I/O PCB	713	166333	233	14037	**W F I
		693	158148	228	13302	**W F I
		16	48000	3000	14989	LGMSG

Compared to the previous run that used 1 TPIPE, the number of IMS waits has dropped significantly, from 159,737 to 1406, whilst the average wait is still approximately 0.2 milliseconds. This suggests that the workload is potentially constrained by the MPRs.

Checking the information reported by the “RESET QSTATS” command, for both commit mode 0 and commit mode 1, the HIQDEPTH is now 3 for each of the two IMS Bridge Queues and 2 for each of the reply queues.

In summary, for Commit Mode 0 there has been a 5.6% increase in the transaction rate simply by using a second TPIPE. The MPRs are not running at capacity, which suggests that using further TPIPs will help drive the workload faster.

For Commit Mode 1, simply by using a second TPIPE, there has been 9.8% increase in the throughput. Since the MPRs are now nearing 100% occupancy, the next step would be to add a further MPR and re-evaluate.

Chapter 10

Hardware Considerations

CPU cost calculations on other System z hardware

CPU costs can be translated from a measured system to the target system on a different zSeries machine by using Large Systems Performance Reference (LSPR) tables. These are available at: [Large Systems Performance Reference for IBM Z](#).

The LSPR workload is now calculated on a workload's relative nest intensity. For general IBM MQ workload, it is recommended to use the AVERAGE relative nest intensity value as this is similar to the previous mixed workload levels and is expected to represent the majority of production workloads.

This example shows how to estimate the CPU cost for a IBM z15 8561-708 where the measurement results are for a 3906-705 (IBM z14):

- The LSPR gives the 3906-705 an average relative nest intensity (RNI) of 14.44
- As the 3906-705 is a 5-way processor, the single engine RNI is $14.44 / 5 = 2.888$
- The “average” RNI of the target 8561-708 used for the measurement is 24.97.
 - The 8561-708 is a 8-way processor.
 - Its single engine ITR is $24.97 / 8 = 3.121$
- The $3906-705 / 8561-708$ single engine ratio is $2.888 / 3.121 = 0.92$ approximately
 - This means that a single engine of a 8561-708 is approximately 8% more powerful than a single engine of a 3906-705.
- Take a CPU cost of interest from this report, e.g. x CPU microseconds (3906-705) per message, then the equivalent on a 8561-708 will be $x * 0.92$ CPU microseconds per message
- To calculate CPU busy, calculate using the number of processors multiplied either by 1,000 (milliseconds) or 1,000,000 (microseconds) to find the available CPU time per elapsed second.

Example: A 3906-705 has 5 processors so has 5,000 milliseconds CPU time available for every elapsed second. So, for a CPU cost of interest from the report of 640 milliseconds on a 3906-705, the CPU busy would be: $640 / (5*1000) * 100$ (to calculate as a percentage) = 12.8%

Caveat

Such CPU cost calculations are useful for estimating purposes. However, extrapolation of throughput potential from one system to another is much more risky because limitations other than CPU power may easily become significant. For instance:

- Inadequate BUFFERPOOL size could mean that page set I/O becomes the limiting factor.
- For persistent messages IBM MQ log data rate is often the limiting factor.
- There may be other limitations beyond the scope of IBM MQ whatever the message persistence used. For instance,
 - Network bandwidth when transmitting messages or using IBM MQ thin clients.
 - You also need to factor in all business logic costs and constraints as there are none in our workloads.

Example: LSPR compared to actual results

An initial set of measurements were run by the IBM MQ Performance group comparing MQ workload on a IBM z15 with workload on an IBM z14 that has been configured in a similar manner.

The following section provides detail of those measurements.

For the set of measurements completed, the z15 out-performed the expectations that were set based on the z14 numbers and the data obtained from the LSPR charts available from:

[Large Systems Performance Reference for IBM Z](#).

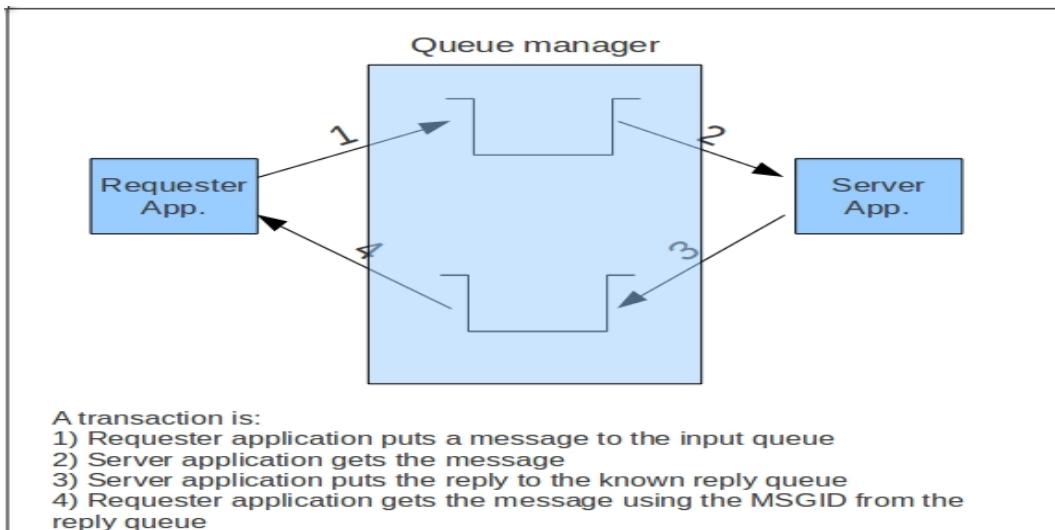
In order to ensure that the tests run on both platforms were directly comparable, only CPU bound tests were run. In conjunction with the relative simplicity of the applications in use, this means that the actual results obtained on the z15 were significantly better than the LSPR may suggest.

Since these tests were CPU bound and the LSPR measurements are based on a mixed type of workload, it is still the recommendation that for production MQ workload estimations, the **average** relative nest intensity (RNI) value is used.

Overview of Environment: Workload

- A set of batch measurements were run against a single WebSphere MQ version 9.1.4 queue manager.
- Only private queues were used.
 - The set of tests incorporated a request/reply model. This took the form shown in the diagram below. Tests were run using 2KB non-persistent messages, with an increasing number of queue pairs in use.
 - For each queue pair in use, there was 1 batch requester and 1 batch server task.

Request / Reply model



Batch Applications

The batch applications used for the measurements are relatively simple and do not include any business logic.

- The requester application is written in PL/X
- The server application is written in C
- All applications are self-monitoring and determine their costs of interest accordingly.

Hardware systems under test

IBM z14 (3906-7H1)

- LPAR with 8 dedicated CPs – 3906 708 (LSPR)

IBM z15 (8561-7G1)

- LPAR with 8 dedicated CPs – 8561 708 (LSPR)

LSPR tables

The information below is an extract of the relevant machines' information from the LSPR website for z/OS v2r3 for “[IBM processors running multiple z/OS images](#)”.

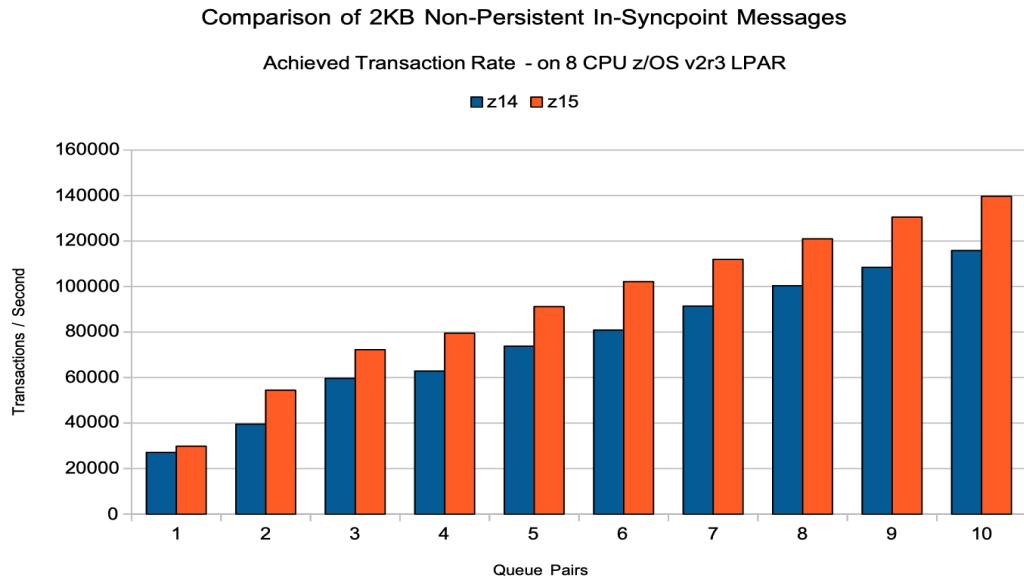
Machine	Processor	#CP	PCI	MSU	Low	Average	High
IBM z14	3906-708	8	12283	1487	24.21	21.91	19.18
IBM z15	8561-708	8	13980	1687	27	24.97	22.19

Non-persistent in-syncpoint messages

This scenario used 1 requester application per request queue that put a non-persistent in-syncpoint message that is subsequently being served by 1 application that gets the message and puts a reply message to a pre-opened queue. This reply message is then gotten by MSGID by the original requester task.

As the workload progresses, the test increments the number of queue pairs in use by 1 until there are 10 pairs in use.

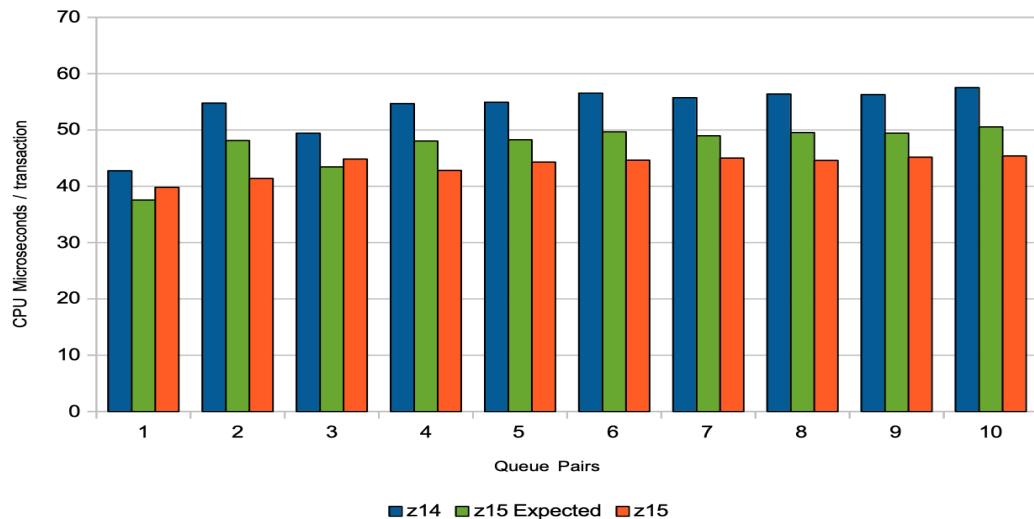
The following chart shows the achieved transaction rates on the z14 and the z15 under test for this scalability benchmark test.



Note: In the type of measurement shown in the previous chart, the IBM z15 has processed the transactions between 10% and 37% faster than the equivalent z14, for an overall average of 23%.

The following chart shows the average cost per transaction based on the achieved (or external) transaction rate and compares it to the expected costs calculated using the LSPR numbers for the machines under test and the algorithm detailed in the section “[CPU Cost calculations on other System Z hardware](#)”.

Actual vs Expected Costs of 2KB Non-Persistent In-Syncpoint Workload



Note: In this previous chart, it can be seen that processing the 2KB workload with 5 queue pairs, the transaction cost on z14 was 55 microseconds, whereas on z15 the similar configuration saw a transaction cost of 44 microseconds, with the predicted cost of 48 microseconds.

Chapter 11

MQ Performance Blogs

MQ Performance Blogs and white papers

The [IBM Integration Community](#) and [Middleware User Community](#) websites are good places to look for some of the latest performance information, but we recognise that sometimes it can be difficult to find the particular item you may be interested in. The purpose of this section is to provide a hint to the performance blogs that might be of interest on the when using IBM MQ on the z/OS platform.

With [MQ for z/OS 9.3](#) introducing the ability to capture statistics and accounting data on separate intervals that are more frequent than previous releases, a short series of blogs discussing SMF options and costs is available:

- “[MQ and SMF - Why, which and how?](#)” discusses why you might want to collect the SMF data, how to enable data collection and which SMF destination to choose.
- “[MQ and SMF - What, when and how much?](#)” discusses the costs of enabling MQ statistics and accounting data collection.
- “[MQ and SMF - How might I process the data?](#)” discusses how you might process the potentially large volumes of SMF data.

Additionally the following blogs may be of interest:

- [Page set performance and best practice](#) discusses the configuration of page sets, how MQ uses page sets and how you can monitor the usage.
- [Moving messages between IBM MQ for z/OS queue managers in a sysplex](#) discusses the configuration options and the performance of those configurations when moving messages between z/OS queue managers in a sysplex. The configurations include Sender/Receiver channels using TCP/IP, SMC-R and SMC-D, as well as the performance of Intra-Group Queuing and using Shared Queues.
- The cost of an MQCONN is relatively high, so in the following blog we discuss some of the factors in that cost: [The cost of connecting to a z/OS queue manager](#).
- Message selection using message properties, which is of interest particularly when using JMS applications to access MQ messages, can be impacted with the use of inefficient message selectors - the blog “[Message selector performance](#)” discusses the best ways to achieve good performance, particularly when accessing messages on shared queues.

- IBM announced that support for data set encryption (DSE) was available from z/OS V2.2 and there are some benefits when using MQ. The blog [MQ and the use of DSE for IBM z/OS v2.2](#) offers some guidance to the cost of encrypting your archive logs. Subsequently in MQ V914 CD, MQ has implemented support for data set encryption for active logs and page sets, and the blog [MQ for z/OS V914 CDR - Data set encryption](#) discusses the performance of this support.
- With security becoming more important, we have created a blog that discusses the [impact of TLS ciphers on MQ channels](#) and offers some guidance as to how the latency might increase when protecting your MQ channels.
- We have also created a blog that discusses [“Impact of certificate key-size on TLS-protected MQ channels”](#) and suggests when you may require additional cryptographic hardware.
- Curious about Dedicated Memory, introduced in z/OS 3.1, and how it might affect your applications (including MQ)? See [“z/OS 3.1, Dedicated Memory and MQ for z/OS”](#).

Blogs may be added at a higher frequency than MP16 updates, so make sure you check back at the [IBM Integration Community](#) and [Middleware User Community](#) websites.

For MQ for z/OS performance white papers, check the [Repository for MQ performance documents](#), which has recently been updated to include:

- Our performance evaluation of the IBM z16 - [“MQ for z/OS on z16”](#).
- Guidance on when to use MQ channel compression as well as the potential benefits - [Channel compression on MQ for z/OS](#).