Performance report – SupportPac MP1J
WebSphere MQ Version 8.0 for z/OS

Version 1.0.1

WebSphere 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.

Second edition, January 2015.

This edition applies to WebSphere MQ for z/OS version 8.0.0 (and to all subsequent releases and modifications until otherwise indicated in new editions).

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 **WebSphere MQ for z/OS V8.0.0**. The information is not intended as the specification of any programming interfaces that are provided by WebSphere 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 WebSphere MQ.

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®
- zEnterprise®
- MQSeries®
- CICS®
- DB2 for z/OS®
- IMS™
- MVS™
- FICON®
- WebSphere®
- InfoSphere®

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
September 2014	Initial Version
January 2015	Minor updates to charts in regression section.

Table of Contents

Performance highlights

This report focuses on performance changes since previous versions, v7.0.1 and v7.1.0, and on the performance of new function in this release.

SupportPac MP16 “Capacity Planning and Tuning for WebSphere MQ for z/OS” will continue to be the repository for ongoing advice and guidance learned as systems increase in power and experience is gained.

Existing function

General statement of regression

CPU costs and throughput are not significantly different in version 8.0 for typical messaging workloads, although there are a number of areas where performance has been improved including:

- Upper bounds of persistent message logging rate
- CICS end of thread processing
- Channel compression.

A user can see how that statement has been determined by reviewing details of the regression test cases in Appendix A.

Storage usage

Virtual storage constraint relief has not been a primary focus of this release, however new function in version 8.0 such as 64-bit buffer pools uses 64-bit storage.

CSA usage

Common Service Area (CSA) storage usage is important as the amount available is restricted by the amount of 31-bit storage available and this is limited to an absolute maximum of 2GB.

The CSA is allocated in all address spaces in an LPAR, so its use reduces the available private storage for all address spaces.

In real terms, the queue manager does not have 2GB of storage to use – as there is some amount used by MVS for system tasks and it is possible for individual customer sites to set the limit even lower.

From the storage remaining of the 2GB of 31-bit storage, a large (but configurable) amount of storage may be used by the queue manager for buffer pools.

The storage remaining is available for actually connecting to the queue manager in a variety of ways and using WebSphere MQ to put and get messages.

Initial CSA usage

CSA usage is similar to V710 when similarly configured queue managers are started.

CSA usage per connection

CSA usage has seen little change in the following releases; V701, V710 and V800.

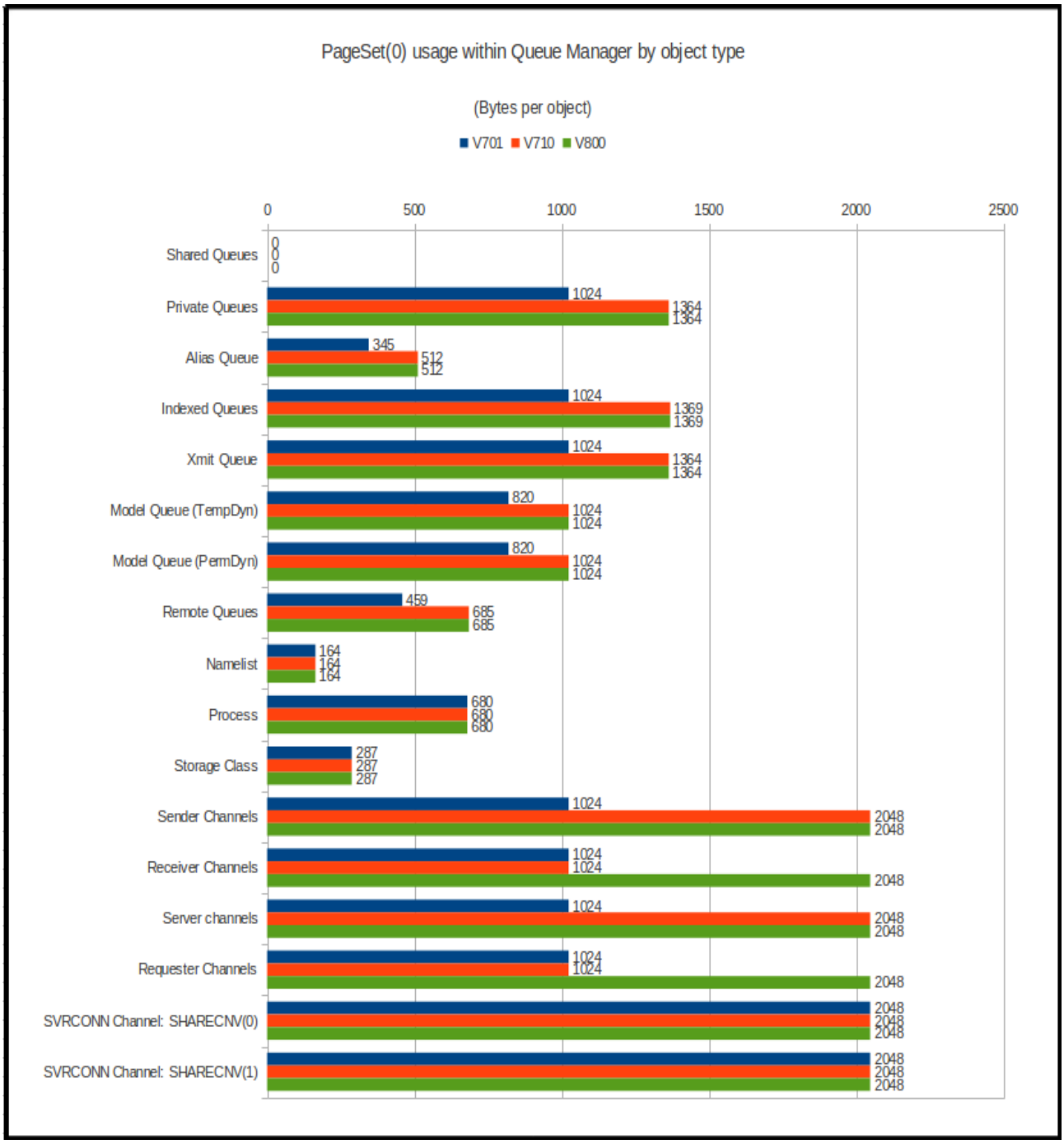
- For local connections, MCA channels and SVRCONN channels with SHARECNV(0), CSA usage is 2.43KB per connection.
- For SVRCONN channels with SHARECNV(1), CSA usage is approximately 4.9KB per connection.
- For SVRCONN channels with SHARECNV(5), CSA usage is approximately 2.9KB per connection (based on 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).

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 on the following 2 charts only includes the storage used when defining the objects.

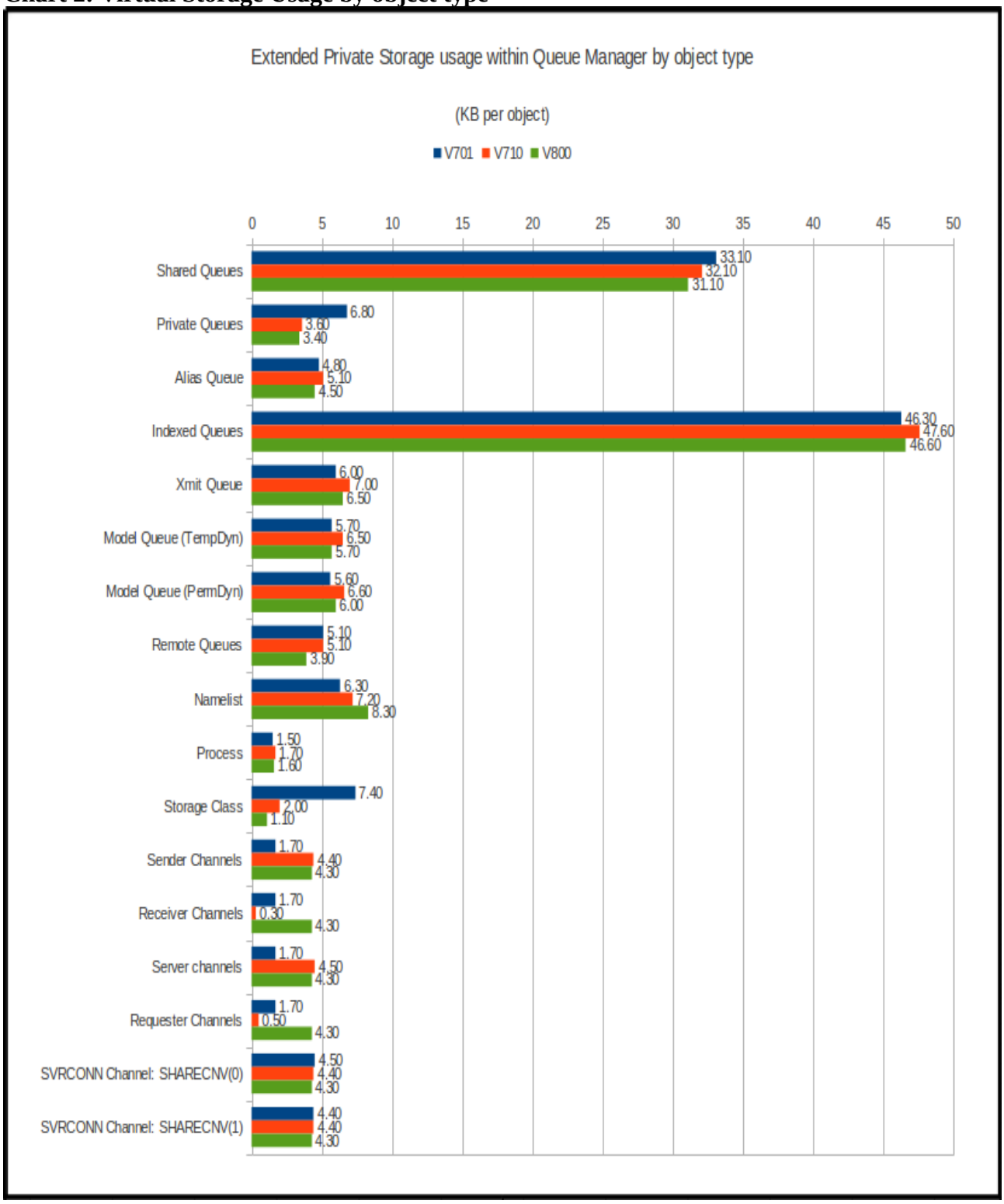
PAGESET(0) Usage

Chart 1: Pageset usage by object type



Virtual Storage Usage

Chart 2: Virtual Storage Usage by object type



Capacity of the queue manager and channel initiator

How much storage does a connection use?

When an application connects to a queue manager, an amount of storage is allocated from the queue manager's available storage.

Some of this storage is held above 2GB, such as security data, and other data is stored on storage taken from that available below the 2GB bar. In the following examples, only allocations from below the 2GB bar are reported.

Previous releases saw typical storage usage is around 22KB per connection. In V800, due to some control areas being moved above the bar, the typical storage usage decreases to 14KB per connection, however there is additional usage in the following (non-exhaustive) cases:

- Where connection is over a SHARECNV(1) channel, the usage increases to 21KB
- Where connection is over a SHARECNV(10) channel and has a CURSHCNV of 10, the usage is 15KB per connection (151KB per channel)
- When connection is over a SHARECNV(1) channel to shared queues - either CFLEVEL(4) or CFLEVEL(5) backed by SMDS, the storage is 28KB, giving an additional shared queue overhead of 7KB

These numbers are based upon the connecting applications accessing a small number of queues. If your application has more than 32 objects open, the amount of storage used will be increased.

If the number of messages held in a unit of work is large and the size of the messages is large then additional lock storage may be required.

How many clients can I connect to my queue manager?

The maximum number of clients you can connect to a queue manager depends on a number of factors, for example:

- Storage available in the queue manager's address space.
- Storage available in the channel initiator's address space.
- How large the messages being put or got are.
- Whether the channel initiator is already running its maximum number of connected clients (either 9,999 or the value specified by channel attributes like MAXCHL).

The table below shows the typical footprint when connecting a client application to a z/OS queue manager via the channel initiator.

The value used in the SHARECNV channel attribute can affect the size of the footprint and consideration as to the setting should be taken. Guidance on the SHARECNV attribute can be found in SupportPacs MP16 “Capacity Planning and Tuning Guide” and MP1F “WebSphere MQ for z/OS V7.0 Performance” report.

		Channel initiator footprint (KB / SVRCONN channel) Message size (KB)			
WebSphere MQ release	**SHARECNV**	**1**	**10**	**32**	**64**
V701	0	98	113	176	207
V710	0	93	106	168	197
V800	0	92	105	168	197
V701	1	179	200	290	353
V710	1	176	197	285	349
V800	1	177	199	288	351
V701	10	269	295	704	1050
V710	10	257	286	695	1038
V800	10	257	285	695	1036

NOTE: For the SHARECNV(10) channels measurements, the channels are running with 10 conversations per channel instance, so the cost per conversation would be the value in the table divided by 10.

How many channels *can* I run to or from my queue manager?

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	1MB
Footprint per channel (channel initiator)	107	115	123	1138
Overhead of message size increase on 1KB messages		+8K	+16K	+1031K

New function for version 8.0

This release has introduced a number of items that can affect performance and these include:

1. 64-bit buffer pools
2. Channel initiator accounting and statistics
3. Multiple cluster transmit queues
4. Message suppression on z/OS using EXCLMSG
5. Advanced Message Security – ***discussed separately.***

In addition there have been a number of items that can impact the capacity of the queue manager

1. 8 byte log RBA
2. 4GB logs

Finally, with the latest hardware supporting zEnterprise Data Compression (zEDC) on z/OS, WebSphere MQ version 8.0 has been updated to be able to exploit the zEDC with channel compression.

64-bit buffer pools

Why use 64-bit buffers?

There are 3 reasons for using 64-bit buffers:

1. Make 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.
2. Enable larger buffer pools to reduce the need for writing to page set and being impacted by the increased I/O time to read/write from page set.
    - This has the additional benefit of reducing the load on the disk subsystem.
3. With the increase to the number of buffer pools, there can be a 1:1 mapping of buffer pools to page sets, so the buffer pools can be more finely tuned to the usage.

As part of the changes to provide the capacity for 64-bit buffers, all of the supporting structures were moved into 64-bit storage[1].

Further to the 64-bit buffer pool changes, there have been improvements to the MQ code to reduce the impact of reading from page set (read-ahead[2]) and asynchronous writes to page set via the deferred write process (DWP).

The deferred write process has been re-written in version 8.0 to allow page set writes to be performed in parallel and to allow more data to be written per request. This increased rate of writing to page set means that there is a reduced likelihood of an MQPUT being stalled whilst the queue manager makes enough room in the buffer pool for the message.

1 These structures include BDSC/hash tables but compared to the buffer pools these are relatively small.
2 Via service parameter

Example: Moving from 31-bit to 64-bit buffer pools

As part of the WebSphere MQ performance tests there is an InfoSphere Replication Server workload which uses 64-bit buffer pools sized at 31-bit buffer pool limits, i.e. 200,000 buffers.

This replication workload simulates moving data from one system to another using MQ channels. The system that sends the data uses a “capture” task to get the data and put to an MQ queue as quickly as possible. At the remote end, there is an “apply” task that gets the messages from the queues and processes them. As the data flows in a single direction, there is the potential for a build up of messages on the transmit queue in that the capture task puts messages more quickly than the channel initiator can get and send the messages, for example in the event of a network delay or the apply task is slow.

Whilst there has been an improvement in the throughput rate from version 7.1 to version 8.0, due in part to DWP, there was also a noticeable increase in the costs associated with the “capture” queue manager, particularly with larger messages.

In these tests the capture queue manager is the one that typically runs short of buffers due to the capture task outpacing the channel initiator get/send task(s). Sampling the workload suggests that the increased cost where small unfixed buffer pools are used is due to the additional I/O operations as MQ reads old messages from pageset and casts out messages from the buffer pool to make room for new messages.

The following tests show the effects of setting the **PAGECLAS(FIXED4KB)** option on the buffer pools where buffer pools are:

- **too small**, sized at 31-bit limits, resulting in asynchronous writes to page set and in the worst cases synchronous writes.
- **large enough** to contain the workload - sized at 4GB.

The workload runs 12GB of data through the system - using either 10KB or 1MB messages in batches of 200 - which we have determined achieves the best throughput on our configuration.

There are 3 sets of tests for each message size:

1. ***Exploit 64-bit storage without increasing size of buffer pool***
   Using 31-bit sized buffer pools located in 64-bit storage if possible, whilst varying the PAGECLAS attribute.
   Queues are allowed to build up to 2GB-worth of data before the channel initiator starts getting messages. As the capture (putter) task is given a head start and is faster than the channel initiator is able to get the message and send across the network, the transmit queue depth continues to increase until page set full occurs and the capture task goes into retry mode.

2. ***Exploit large buffer pools where queue fits entirely in buffer pool***
   Using 4GB buffer pools located in 64-bit storage, whilst varying the page-fixed nature of the buffer pool.
   Channels are active at the start so the queue depths grow more naturally as the capture task puts faster than the channel initiator is able to get the messages until the queues hit page set

full and the task goes into retry mode.

3. ***Exploit large buffer pools when queue may not fit entirely in buffer pool***
   Using 4GB buffer pools located in 64-bit storage, whilst varying page fixing option. Queues are allowed to build up to 2GB-worth of data before the channel initiator starts getting messages. As the capture (putter) task is given a head start and is faster than the channel initiator is able to get the message and send across the network, the transmit queue depth continues to increase until page set full occurs and the capture task goes into retry mode.

The costs shown in following 2 charts are the CPU costs per message processed by the capture queue manager address space.

Chart 3: Using 10KB messages

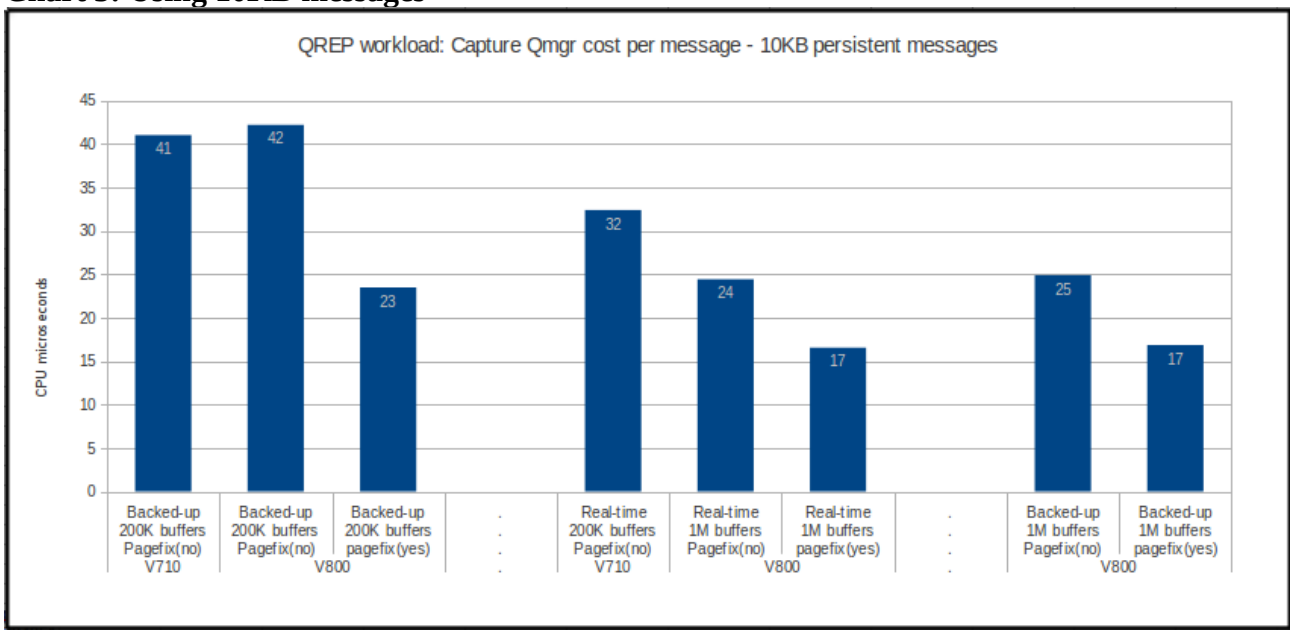
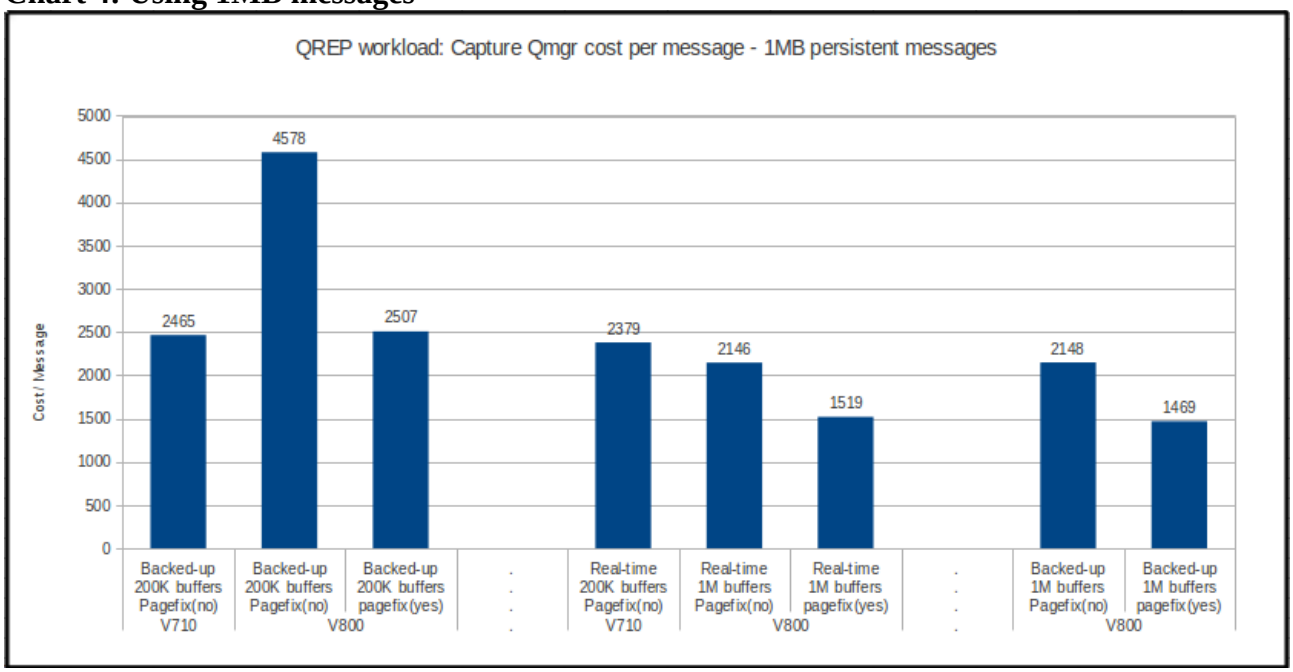


Chart 4: Using 1MB messages



Notes on charts 3 and 4:

- The throughput rate for version 8.0 has not been significantly affected by page-fixing the buffers, however the tests are configured to have sufficient CPU at all times.
- Only the capture queue manager costs varied by any significant value.
- Moving from version 7.1 to version 8.0 with PAGECLAS(4KB) saw an increase in capture queue manager cost, particularly for 1MB messages.
  - Specifying PAGECLAS(FIXED4KB) brought costs back in-line with version 7.1 for large messages (reducing version 8.0 costs by ~36%) and significantly reduced the 10KB message costs (~48%).
  - Comparing the first and second sets of tests show that increasing the size of the buffer pool brings the costs below those observed when using 31-bit buffer pools.
- When larger buffer pools were used (version 8.0 only), PAGECLAS(FIXED4KB) reduced the cost (~30%) for both small and large message workloads.

Why not specify page-fixed as the default?

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

Some customers use long-term storage protection where WLM restricts storage donations to other work. This option can be useful for work that needs to retain storage during long periods of inactivity because it cannot afford paging delays when it becomes active again. With long-term storage protection assigned, this work will lose storage only to other work of equal or greater importance that needs the storage to meet performance goals.[3]

The PAGECLAS attribute determines when the buffer pool pages are fixed.

- 4KB will be page-fixed and released between use.
- FIXED4KB will page-fix the storage at allocation and only release at end of task.

Even on systems where long-term storage protection is enabled, the buffer pools allocated with PAGECLAS(4KB) will be paged-fixed / released as required and in long periods of inactivity may be donated to higher importance work.

Provided these pages have not been donated, the actual pages used *should* remain the same, however there is an inherent cost with page-fix/release, so PAGECLAS(FIXED4KB) can still offer a higher level of storage isolation at reduced cost.

If there is a need for buffers to be allocated in 64 bit storage it is advisable to ensure there is sufficient real memory available such that PAGECLAS(FIXED4KB) can be specified.

3 Long-term storage protection is assigned with the “Storage Critical” option, found by scrolling right on the “Modify Rules for the Subsystem Type” panel in WLM.

Example: Benefits of expanding the buffer pool to contain the workload

When a message is gotten from page set, the queue manager must wait for the read of the page set to complete. Even on a lightly loaded system this can be a significant impact.

For example the following is an extract from data gathered with accounting class(3) enabled and subsequently formatted using CSQW116S and shows the cost of an MQGET by a long running batch task that is randomly selecting and getting a 1000 byte message from an indexed queue.

```
MQ call-          N          ET          CT        Susp        LOGW        PSET
Get     :   1302949          40          10          30           0          28
```

This is telling us that on average of the 1,302,949 gets performed, they cost 10 CPU microseconds but took 40 microseconds to complete. Of the 30 microseconds the task was suspended, it spent 28 microseconds waiting to get the message back from page set.

By contrast the following is an extract from a comparable test where the buffer pool is large enough to contain all of the messages.

```
MQ call-          N          ET          CT        Susp        LOGW        PSET
Get     :   2334042          13          11           2           0           0
```

In this instance, the same 79 second interval achieved 2,334,042 gets (80% more) at the slightly higher cost of 11 CPU microseconds but the task was only suspended for 2 microseconds – and never for page set.

Example: Benefits of larger buffer pools for transmission queues

In many ways this example is an extension of the previous “Benefits of expanding the buffer pool to contain the workload”, but offers more detail.

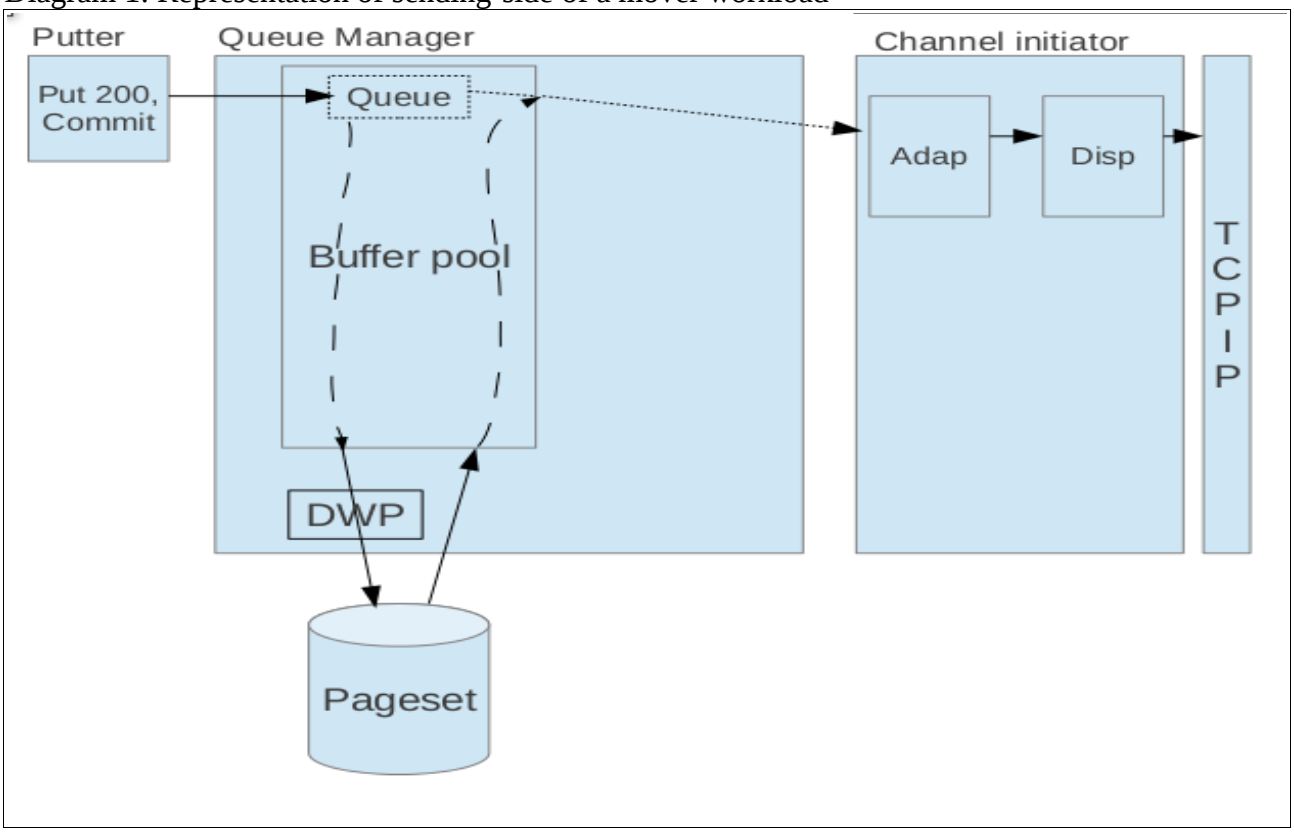
The test moves 2KB non-persistent messages between queue managers over a single sender-receiver channel pair but the example will concentrate only on the sending-side of the workload.

There are 2 configurations shown;

1. Insufficient buffers to contain the workload - using 31-bit buffer pools with 200,000 buffers.
2. Sufficient buffers to contain the workload - using 64-bit buffer pools with 1,048,576 buffers

The following diagram describes the flow that a message may take between the putting task generating the message to actually sending the message across the network.

Diagram 1: Representation of sending-side of a mover workload



Notes on diagram 1:

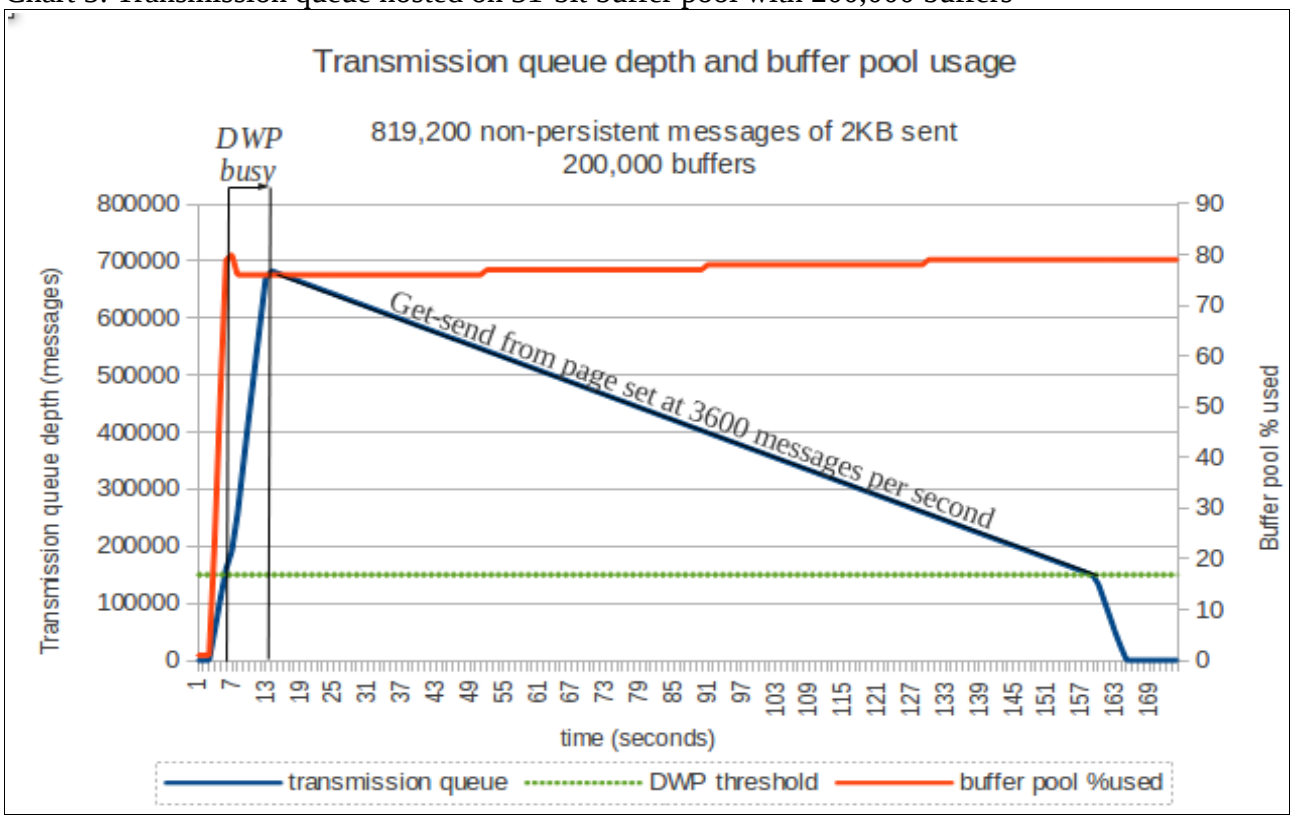
- The putting task is putting 200 messages of 2KB to the remote queue and then issuing a commit. This process is repeated until the batch application has put over 800,000 messages.
- The messages are being got from the queue by the adaptor task and put onto the network by the dispatcher task. This pairing of getting and putting onto network by adaptor and dispatcher task will be termed “get-send”.
- Even with channel batch sizes of 200, the network latency means that the putting task is out-

pacing the channel initiator adaptor (getter) task and the queue depth is increasing.

- As the queue depth increases, the buffer pool usage does too. As this begins to fill, the deferred write processor task (DWP) starts to cast out the messages from buffer pool to page set, to make room for newer messages.
- This deferred write task runs asynchronously as an SRB. Should the DWP be unable to keep enough buffers available in the buffer pool, the writing of buffers to page set must be performed synchronously by the putting task.
- Once the messages required by the adaptor task are on page set, the queue manager uses the getting task to read the page set. Page set I/O is typically much slower than reading directly from buffers and the performance can be impacted by poor disk response times. This getting from page set is done using a TCB and will be charged to the application – in this case the channel initiator.

Configuration 1: Insufficient buffers to contain the workload – 200,000 buffers.
The following chart plots the depth of the transmission queue with the buffer pool usage, as determined by a separate application using PCF messages to query the data on a per second basis.

Chart 5: Transmission queue hosted on 31-bit buffer pool with 200,000 buffers



Notes on chart 5:

- The workload starts approximately 4 seconds after the monitoring begins.
- The queue depth grows very quickly as the putting task out-paces the channel initiators get-send task.
- The DWP threshold *(y1-axis)* is based on 75% of the 200,000 buffers being used – i.e. 150,000 buffers (and by implication 150,000 messages. since there is 1 message per 4K

buffer). DWP actually begins when 85% of the buffers are used and then aims to keep the buffer pool usage between 75-85% used.

- Within 6 seconds of the measurement starting, the buffer pool usage had exceeded the DWP threshold. From that point on, the DWP SRB was busy writing data to page set to ensure sufficient space in the buffer pool in order to avoid impacting the applications putting the messages. This continued until the putter task completed. At this point DWP did not need to do any more work but even with the application checking the buffer pool usage on a per second basis, we could not determine whether the threshold to stop DWP had been reached.
- The putter task finished at 14 seconds, at which point the queue was at its peak depth of 685,542 messages.
- Once the putter completes, the channel initiator's get-send process becomes the dominant factor in the queue depth, i.e. the depth decreases. There is a marked difference between the rate of get from page set and buffer pool. This can be seen by the angle of the line showing queue depth dropping – in particular the rate from 14 seconds to 160 seconds where the messages are being retrieved from page set at a rate of approximately 3600 messages per second, compared to the rate of gets from buffer pool shown at 160 seconds onwards where the get rate is 28,000 messages per second.

Digging a bit deeper...
Using both class 3 and class 4 accounting and statistics data we can look at the separate tasks and determine the cost of the workload. These different accounting classes give different information, class 3 giving task related data once inside the MQ API and class 4 giving channel initiator task related.

Class 3 accounting: SMF type 116 records are extracted and then formatted using CSQW116S.

Putter task:

```
== Interval   : START 10-07-2014 20:13:14.52
== Interval   : END   10-07-2014 20:13:24.67
..
== Commit     : Count       4096,  Avg elapsed         82,  Avg CPU          5
..
Page set ID             1,  Buffer pool          1
..
PUTs: Valid    819200,  Max size      2048,  Min size      2048,  Total bytes   1600 MiB
-MQ call-          N         ET         CT       Susp       LOGW       PSET
 Open   :          1         18         18          0
 Close  :          1          3          3          0
 Put    :     819200          9          7          1          0
..
Maximum depth encountered     685542
```

This data shows there were 819,200 MQPUTs each of 2048 bytes that cost an average of 7 CPU microseconds and took on average 9 microseconds of elapsed time.

For those 819,200 MQPUTs there were 4096 commits – which gives an average of 200 messages per commit.

Channel initiator "get-send" task:
The class 3 accounting records span 2 intervals. The first interval commences when the channel

starts and contains the following API data:

```
..
Page set ID              0,  Buffer pool          0
..
GETs: Valid    492310,  Max size      2476,  Min size      2476,  Total bytes 1162 MiB
GETs: Dest-S        0,  Dest-G    492311,  .. ,          Successful destructive 492310
..
-MQ call-           N         ET         CT       Susp       LOGW       PSET
 Open    :          1          4          4          0
 Get     :     492311        196         15          0          0          0
 Inquire:           1          3          3
```

In this interval we can see that the get cost an average of 15 microseconds per request but took 196 microseconds – a delay of 181. This lengthy delay is waiting for the data to be retrieved from page set.

Note: There is a defect that incorrectly identifies which page set is associated with the transmission queue on the first open of the queue. As the channel initiator opened the queue before the putting application, the page set ID is recorded as “0”. As a result of this, the time spent waiting for page set is not recorded.

The second interval includes the gets directly from buffers but there are also many gets which involve page set I/O.

```
..
-MQ call-           N         ET         CT       Susp       LOGW       PSET
 Get     :     326891        141         13          0          0          0
```

In the second interval the average CPU cost of the MQGET decreases slightly to 13 microseconds and the elapsed time has dropped from 196 to 141 microseconds.

Class 4 statistics:
In WebSphere MQ version 8.0 for z/OS, there are the additional options of class 4 accounting and statistics which do have some overlap with the class 3 accounting discussed above. As this workload is well defined and for simplification purposes, only the adaptor and dispatcher reports are discussed.

Adaptor report:
The following table is an extract of the data from statistics class(4) adaptor data and only shows the relevant information for this example.

Interval	Number of requests	Busy %	CPU %	Average CPU (microseconds)	Average Elapsed (microseconds)
1	268818	68.9	8.1	18	154
2	223543	94.7	7.9	21	254
3	326892	78.9	9.4	17	145
Total	819253			*15090291*	*145577234*
Average				*18.42*	*177.7*

Note the average CPU time ranges from 17 to 21 microseconds, which is slightly higher than that seen in the class 3 accounting data for the channel initiator getter task. This is due to the accounting data being measured in slightly different places and having different intervals.

The “Busy %” describes how much of the interval the adaptor is processing API requests. Interval 2 shows the adaptor task being 94.7% busy for the 60 second interval, despite the CPU being only 7.9% busy. This shows that the adaptor is blocked waiting for something – in this case it is waiting for the page set IO to complete. It also suggests that to support multiple channels, more than 1 adaptor may help performance.

The total number of adaptor requests is approximately equal to the number of messages sent over the channel.

Dispatcher report:
The following table is an extract of the data from dispatcher statistics data and only shows the relevant information for this example.

Interval	Number of requests	Busy %	CPU %	Average CPU (microseconds)	Average Elapsed (microseconds)
1	538928	4.5	5.2	6	6
2	448204	4.1	4.7	6	6
3	655420	5.5	6.3	6	6
Total	1642552			*9855312*	*9855312*
Average				*6*	*6*

In this example the total number of dispatcher requests is approximately double the number of messages. There is typically 1 request per message plus 1 request per message segment. Each message is processed in segments of up to 32KB, although the segments may be smaller on SSL encrypted channels.

The average elapsed time may show a value less than the average CPU time was the TCBTIME field is only updated when the TCB calls into the z/OS dispatcher leading to an underestimate of CPU time.

In this instance, the dispatcher being used is not busy and can support more channels.

How do the adaptor and dispatcher costs relate to the data in the RMF CPU report?
In the above section, we have determined that the total cost of the adaptor was 15.1 CPU seconds and the total cost of the dispatcher was 9.8 CPU seconds, giving a total cost of 24.9 CPU seconds.

According to the RMF CPU report for the same period, the channel initiator used 26.548 CPU seconds (TCB plus SRB).

This 6% difference suggests that the channel initiator was doing little other work.

DWP costs
Given the buffer pool usage meant that the DWP task was required and the DWP runs as an SRB within the queue manager, we can calculate an approximate cost using the RMF CPU report. In this the total SRB time was 2.64 CPU seconds, or 3.22 microseconds per message[4].

Cost of sending the messages – using the class 4 statistics data where possible
Now that we have the costs from the channel initiator, we can calculate the cost of sending each message:

Task	**CPU Time**	**Elapsed Time**
Application (putting)	7	9
Queue Manager: DWP	3.22	3.22
Channel Initiator: Adaptor	18.42	177.7
Channel Initiator: Dispatcher *(2 requests per message)*	12	10
Total	40.64	199.92

In this example, nearly 80% of the elapsed time is spent waiting and those waits are primarily in the adaptor task waiting for the message to be read from page set.

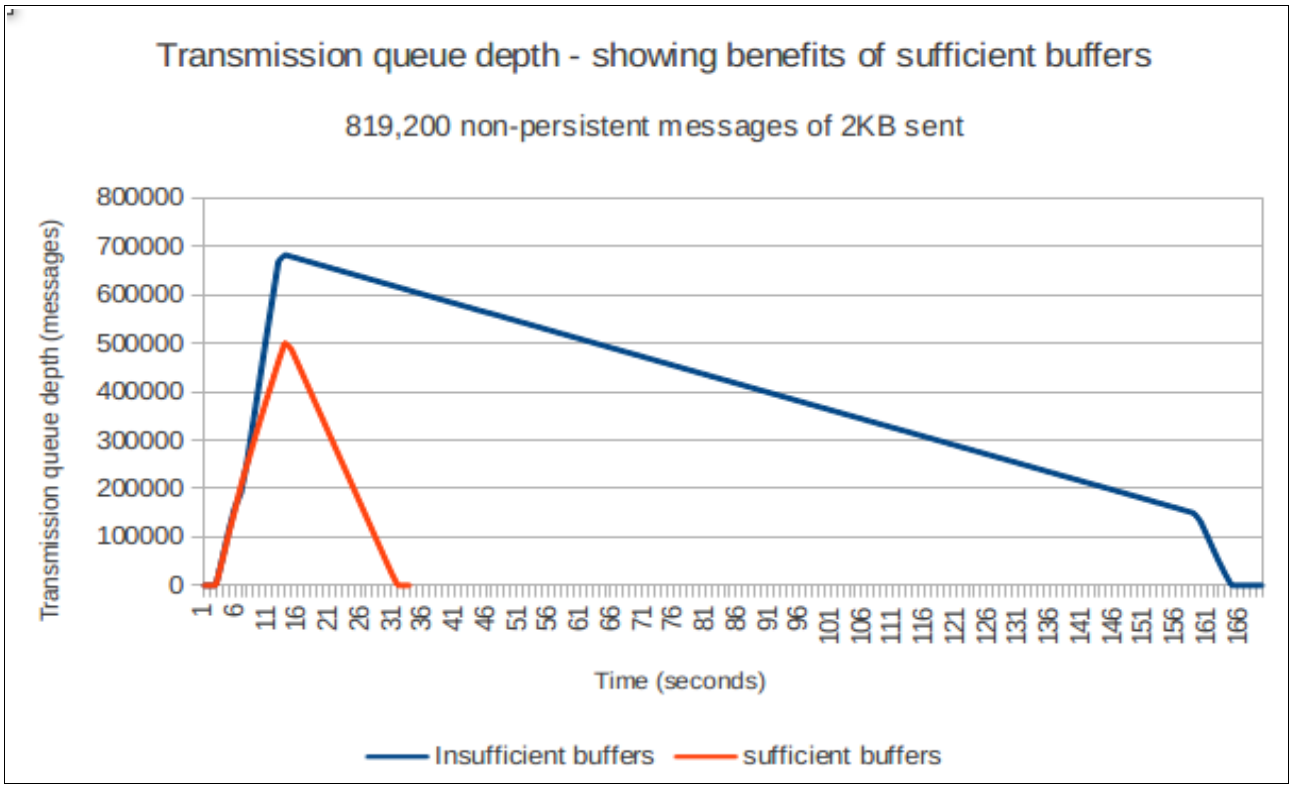
4 In this example the queue manager is only running this workload so there is little other cost to be excluded from the CPU report.

Configuration 2: Sufficient buffers to contain the workload – 1,048,576 buffers.

In this configuration, the queue manager is configured with a 4GB of buffer pool which will be used by the sending-side transmission queue. The buffers are not page-fixed as the workload fits within the buffer and the system has sufficient storage that paging will not occur. In systems where the workload may not fit or paging may occur then some benefit may be seen when using PAGECLAS(FIXED4KB).

The following chart shows the transmission queue depth seen previously in chart 5 when using a buffer pool limited to 200,000 buffers and compares it when the buffer pool exploits 64-bit storage and uses a 4GB buffer pool.

Chart 6: Comparing transmission queue depth



Notes on chart 6:

- The time from start to end of workload decreases from 166 seconds to 33 seconds. This means that the transfer rate has increased from 9.6 to 48.5 MB per second.
- The queue depth when running with sufficient buffers hits a peak of 510,695, which is 175,000 lower than when there are not enough buffers.
- The putting task completes within 14 seconds in both measurements.
- The rate of get-send by the channel initiator is consistent and is approximately 29000 messages per second.

Digging a bit deeper...
In this example we will rely primarily on the class 4 statistics data for the channel initiator tasks as this shows more complete information for the tasks being used.

Class 3 accounting: SMF type 116 records are extracted and then formatted using CSQW116S.

Putter task:

```
-MQ call-          N         ET         CT       Susp       LOGW       PSET
 Open    :         1         18         18          0
 Close   :         1          5          5          0
 Put     :    819200         10          9          0          0
```

Note, compared to the 31-bit buffer pool measurement there is an increase in both CPU (+2 microseconds) and elapsed time (+1 microsecond) per put.

This increased cost however is offset by the buffer pool never being more than 50% full[5], so the DWP task is not busy. Furthermore using the RMF CPU report for the intervals that the workload ran confirms that the queue manager used less SRB CPU time - only 0.276 CPU, which equates to 0.33 CPU microseconds per message.

Class 4 statistics:
As before, this section will only consider the adaptor and dispatcher tasks within the class 4 statistics data.

Adaptor report:
The following table is an extract of the data from adaptor statistics data and only shows the relevant information for this example.

Interval	Number of requests	Busy %	CPU %	Average CPU (microseconds)	Average Elapsed (microseconds)
1	819258	16.9	17.51	13	12
Total	819258			*10650354*	*9831096*

In this case, the workload completed in a single interval, with a much reduced CPU and elapsed time when compared to the measurement with insufficient buffers.

Dispatcher report:
The following table is an extract of the data from dispatcher statistics data and only shows the relevant information for this example.

Interval	Number of requests	Busy %	CPU %	Average CPU (microseconds)	Average Elapsed (microseconds)
1	1642550	12	13.8	5	5
Total	1642550			*8212750*	*8212750*

5 According to the Websphere MQ “Buffer Manager” report, the buffer pool was never more than 50% used with the low watermark at 530661 buffers.

In this example the total number of dispatcher requests is again double the number of messages. There is typically 1 request per message plus 1 request per message segment, so we would need to account for 2 dispatcher requests per message in the total cost per message calculation.

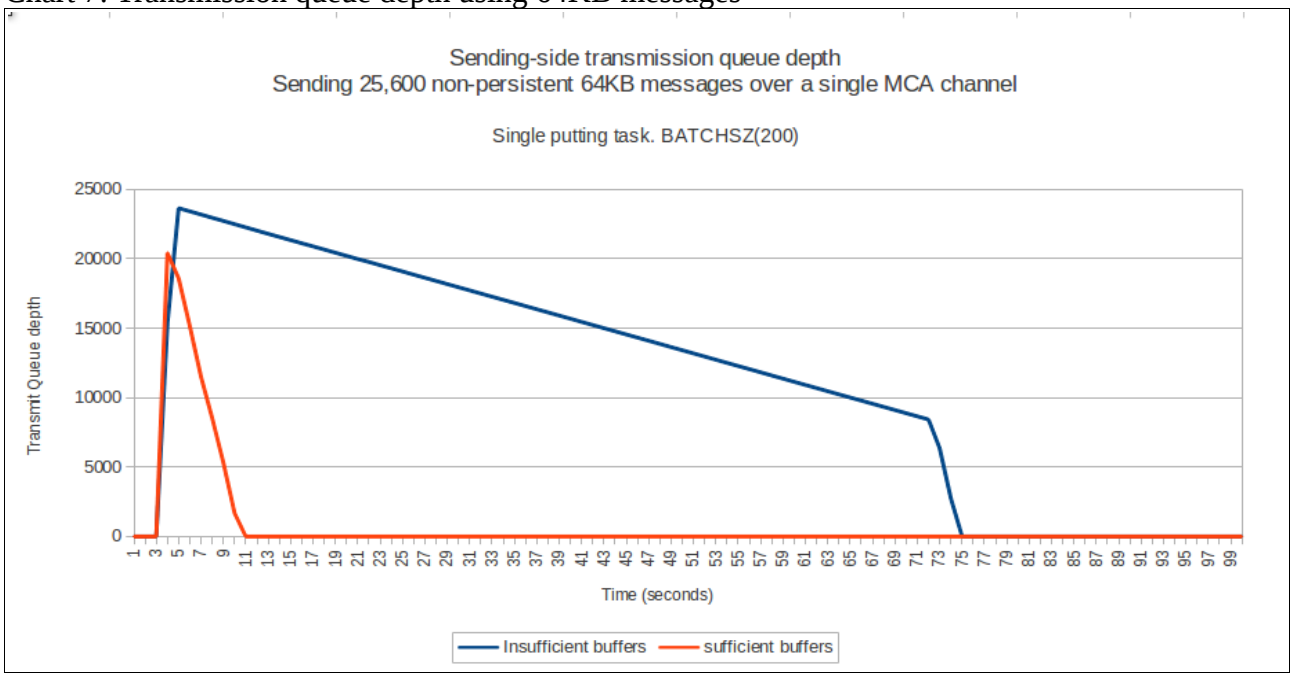
Cost of sending the messages – using the class 4 statistics data where possible.
Now that we have the costs from the channel initiator, we can calculate the cost of sending each message:

Task	**CPU Time**	**Elapsed Time**
Application (putting)	9	10
Queue Manager: DWP	0	0
Channel Initiator: Adaptor	13	12
Channel Initiator: Dispatcher *(2 requests per message)*	10	10
Total	32	32

In the example where the workload fits within the available buffers, thereby avoiding delays due to page set I/O, the costs have been reduced by 25% and the elapsed time has been reduced by 84%.

What about larger messages?
A similar pattern can be seen when using larger messages, for example using 64KB messages gives the following chart of transmission queue depths

Chart 7: Transmission queue depth using 64KB messages



Following the same process as described earlier in the example, we can build the following table of costs:

Task	Insufficient buffers		Sufficient buffers	
	CPU time	Elapsed time	CPU time	Elapsed time
Application	36	61	35	35
DWP	43.83	43.83	0	0
Adaptor	137	2672	68	68
Dispatcher (4 requests per message[6])	41.4	41.4	32	32
Total	258.23	2818.23	135	135

In this case the costs of the workload have reduced by nearly 50% and the elapsed time by 85%.

6 A 64KB message has MQ implementation headers which means that approximately 64.5KB of data will be sent. This means that there are 3 segments per message.

Channel initiator accounting and statistics

The usage of channel initiator accounting and statistics data for reporting and tuning purposes will be discussed in performance report MP1B.

Storage requirements
When a version 8.0.0 channel initiator is started, it will attempt to allocate approximately 190MB above the bar virtual storage for channel initiator accounting and statistics, regardless of whether class 4 trace is enabled.

It is suggested that the channel initiator is allowed access to a minimum of 256MB of virtual storage and this may be set by using specifying **MEMLIMIT=256M**.

If the MEMLIMIT parameter is not set in the channel initiator JCL, the amount of virtual storage above the bar may be set from by the MEMLIMIT parameter in the SMFPRMxx member of SYS1.PARMLIB or from the IEFUSI exit.

If the MEMLIMIT is set to restrict the above bar storage below the required level, the channel initiator will issue the CSQX124E “SMF task ended abnormally” message and class 4 accounting and statistics trace will not be available.

Cost of running with class 4 accounting and statistics trace
Enabling accounting trace class(4) and/or statistics trace class(4) typically impacts the channel initiator CPU usage by between 1 to 2%. For some channels where the cost of processing a message is higher, such as those using encryption or channel compression, the overhead channel initiator accounting and statistics may be less than 1%.

Multiple cluster transmit queues

WebSphere MQ version 8.0 for z/OS supports multiple cluster transmission queues, which was introduced in version 7.5 on distributed platforms.

The use of multiple cluster transmission queues means that:

- Transmission queues can be specified for particular cluster sender channels, providing higher availability as the messages can be isolated from other cluster-sender channels.
- The queue attribute CLCHNAME can be used to group multiple cluster sender channels for a transmission queue.
- Transmission queues can be placed on separate page-sets via storage classes, which when used in conjunction with different buffer pools can allow different class of service. For example one cluster workload may be regarded as a higher class priority and could be placed in a separate buffer pool that is sufficiently large to contain all of the workload. Clustering always does an MQGET by index and the benefits of getting messages from buffer pool rather than page set can be seen here.

Switching to multiple cluster transmit queues

The check to determine whether to use the default cluster transmit queue or multiple cluster transmit queues only occurs when a cluster channel starts or restarts.

If there are messages on the cluster transmit queue being switched from for the newly starting channel, they will be moved from that transmission queue to the required transmission queue. If these are persistent messages then the moving of these messages will be logged, which may increase the time taken to move the messages as it will be affected by the rate the queue manager can log the messages and what other persistent workload is already being logged.

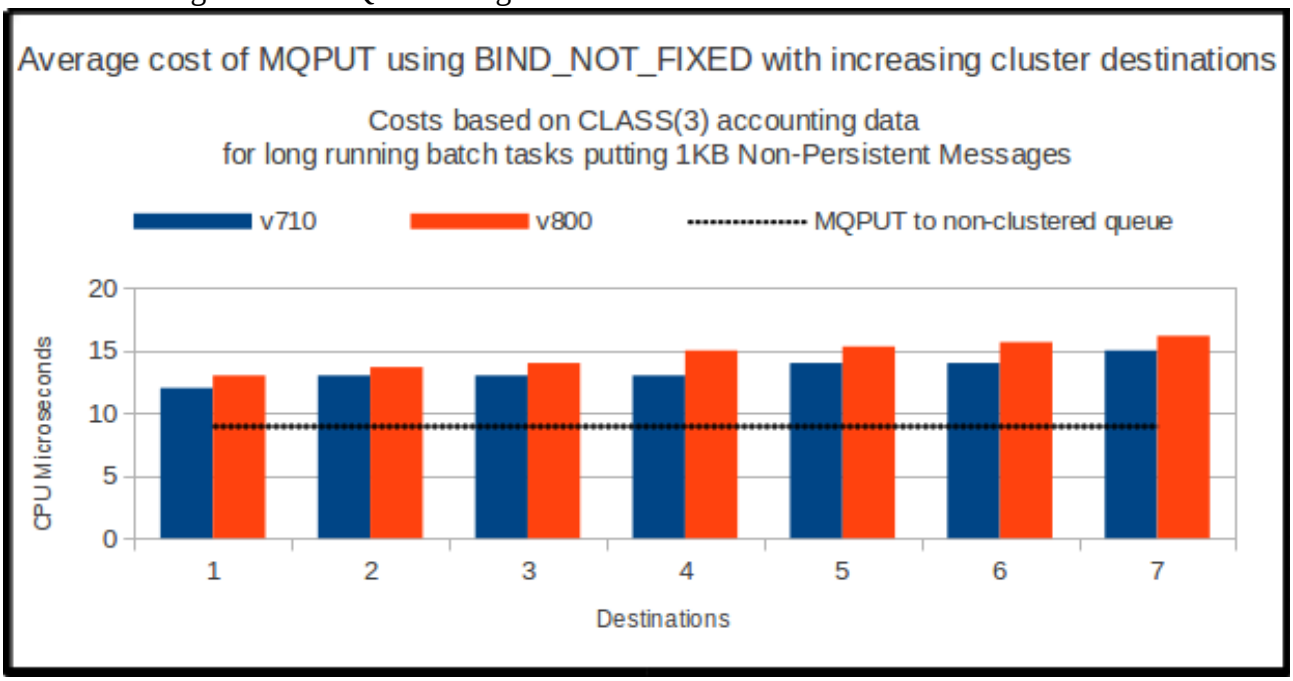
Do multiple cluster transmit queues affect performance?

In order to support multiple cluster transmit queues, the queue manager must determine which transmission queue the message is being put to. This additional processing has increased the cost of the MQPUT to clustered queues in version 8.0.

This additional cost is seen at the MQPUT time regardless of cluster bind option.

The following chart compares the cost of an MQPUT of a 1KB non-persistent message to both non-clustered and clustered queues.

Chart 8: Average cost of MQPUT using Bind-not-fixed



Notes on chart 8:

- The cost of a put to a clustered queue is greater than the cost of a put to a non-clustered queue. There is an inherent cost with clustering which will occur for different MQ API's depending on the bind type:
  - BIND_ON_OPEN will see an increase at MQOPEN cost compared to an open of a non-clustered queue
  - BIND_NOT_FIXED will see an increase in MQPUT cost compared to a put to a non-clustered queue.
  - MQPUT1 will see a similar increase regardless of bind type.
- The cost of the put to clustered queues increases as the number of destinations increase.
- The overhead of supporting multiple cluster transmit queues is between 1 and 2 CPU microseconds and is independent of the size or persistence of the message.
- As can be seen in "Appendix A - moving messages across cluster channels", the overall transaction cost generally has stayed in-line with previous releases of WebSphere MQ.

Message suppression on z/OS using EXCLMSG

In WebSphere MQ version 8.0 for z/OS, it is possible specify a list of error messages to be excluded using a new ZPARM attribute, EXCLMSG.

In the case of frequently occurring messages, for example in an environment where clients connections are starting and ending with high frequency, there are several benefits in excluding the CSQX511I and CSQX512I messages[7]:

1. Reduce the amount of spool space used
2. Reduce the CPU costs from the suppressed message being issued to the channel initiator job output.

Rebuilding the ZPARM using macro CSQ6SYSP and specifying “EXCLMSG=(X511,X512)”, the cost of 200,000 short-lived clients transactions compared favourably against a similarly configured queue manager that did not have error message suppression enabled.

The client transaction issued:

- connect
- open
- put or get of a 1KB non-persistent message
- close
- disconnect

The measurements with messages CSQX511I and CSQX512I suppressed showed a reduction in transaction cost of approximately 20% compared to the run with no message exclusion enabled.

Transactions that perform more MQ API requests during the connection life-time would see a lower percentage of savings from the connect and disconnect phase of their transaction. Similarly transactions connecting with security enabled may also see a reduced cost benefit.

7 The CSQX511I and CSQX512I messages have been introduced for SVRCONN channels to differentiate between normal channels and SVRCONN channels.

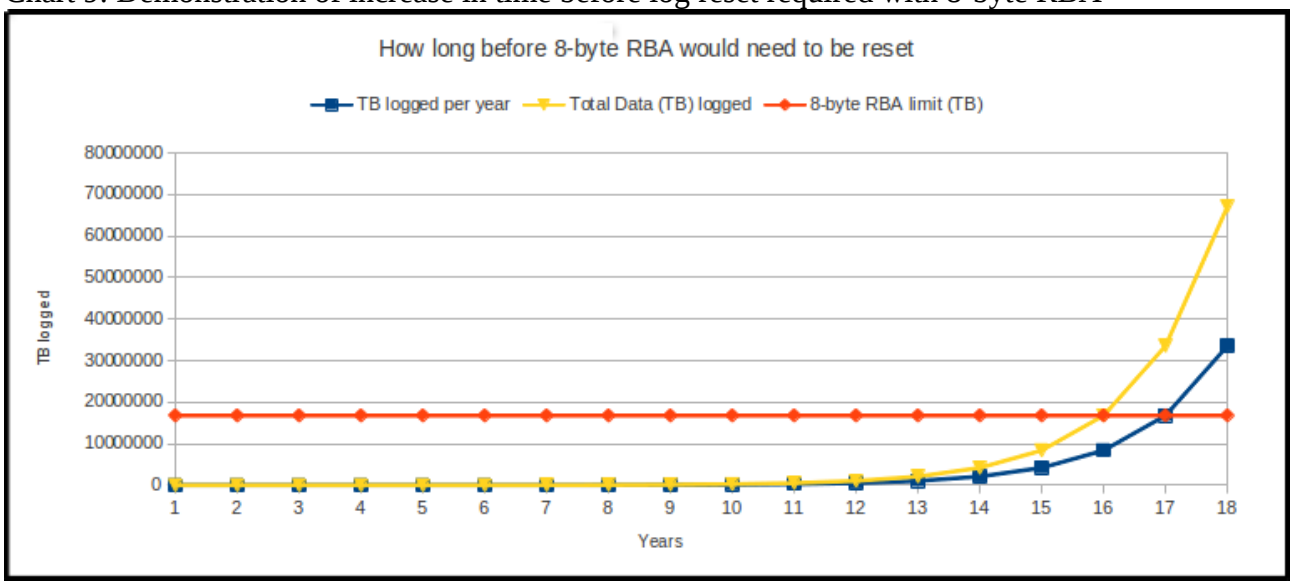
8 byte log RBA

WebSphere MQ Version 8.0 for z/OS improves the availability of the queue manager by increasing the period of time before the log needs to be reset.

In releases prior to version 8.0, the 6 byte log RBA could address up to 256 terabytes of data.
In version 8.0 the log RBA can be 8 bytes long and the queue manager can now address over 64,000 times as much data (16 exabytes) before the log RBA needs to be reset.

This means that in a system that currently logs 256 TB of data annually i.e. would currently be resetting the logs on an annual basis, but anticipates a 100% compound growth rate per annum, using the 8-byte RBA would mean the logs would not need resetting for 17 years, e.g.

Chart 9: Demonstration of increase in time before log reset required with 8-byte RBA



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 were 64KB, additional 2% of log space would be used compared to a queue manager using a 6-byte RBA.

Despite the increase in the amount of data logged, we have found that the upper bounds for log rate for a particular message size are not impacted significantly. Typically we have observed less than 2% difference in persistent messaging rate over a similarly configured version 8.0 queue manager using a 6-byte RBA.

4GB logs

The size of the active log is limited to the lower value of 4GB or the maximum size of the archive dataset.

When archiving to tape, WebSphere MQ has the ability to write to multiple tapes, allowing the use of a 4GB log.

Prior to version 8.0, the maximum size of an archive log stored on DASD was limited by the BDAM access method to 65,535 tracks which is just under 3GB and as such the log should have been sized accordingly.

A queue manager running WebSphere MQ version 7.1 for z/OS defined with 4GB logs and archiving to DASD will appear to archive successfully but if the archives need to be read there will be gaps.

In version 8.0, the archive log access method had been changed from BDAM to BSAM and this means that the size of the archive log written to DASD can be increased to 4GB, and subsequently the active logs can now be increased to 4GB.

Note, as the WebSphere MQ documentation states:

"Care should be taken when defining a log of 4GB as the system may round up the number of records specified to a value that results in a log being greater than 4GB.

When the queue manager reads such a log, the log is seen as being much smaller than it actually is, giving undesired results; for example very frequent log switches.

When defining a large log, you should check the HI-A-RBA value of the log using IDCAMS LISTCAT, to ensure that the log is strictly less than 4GB (4,294,967,296 bytes)."

This means that the queue manager can hold a maximum of 124GB[8] of data in its active logs.

8 This is calculated from 31 active logs each of 4GB.

Channel compression using zEDC

Channel compression typically is used on high latency / low band-width networks to reduce the amount of traffic being flowed as it has the effect of increasing CPU costs per message.

The latest class of System z hardware introduces hardware compression via zEnterprise Data Compression (zEDC).

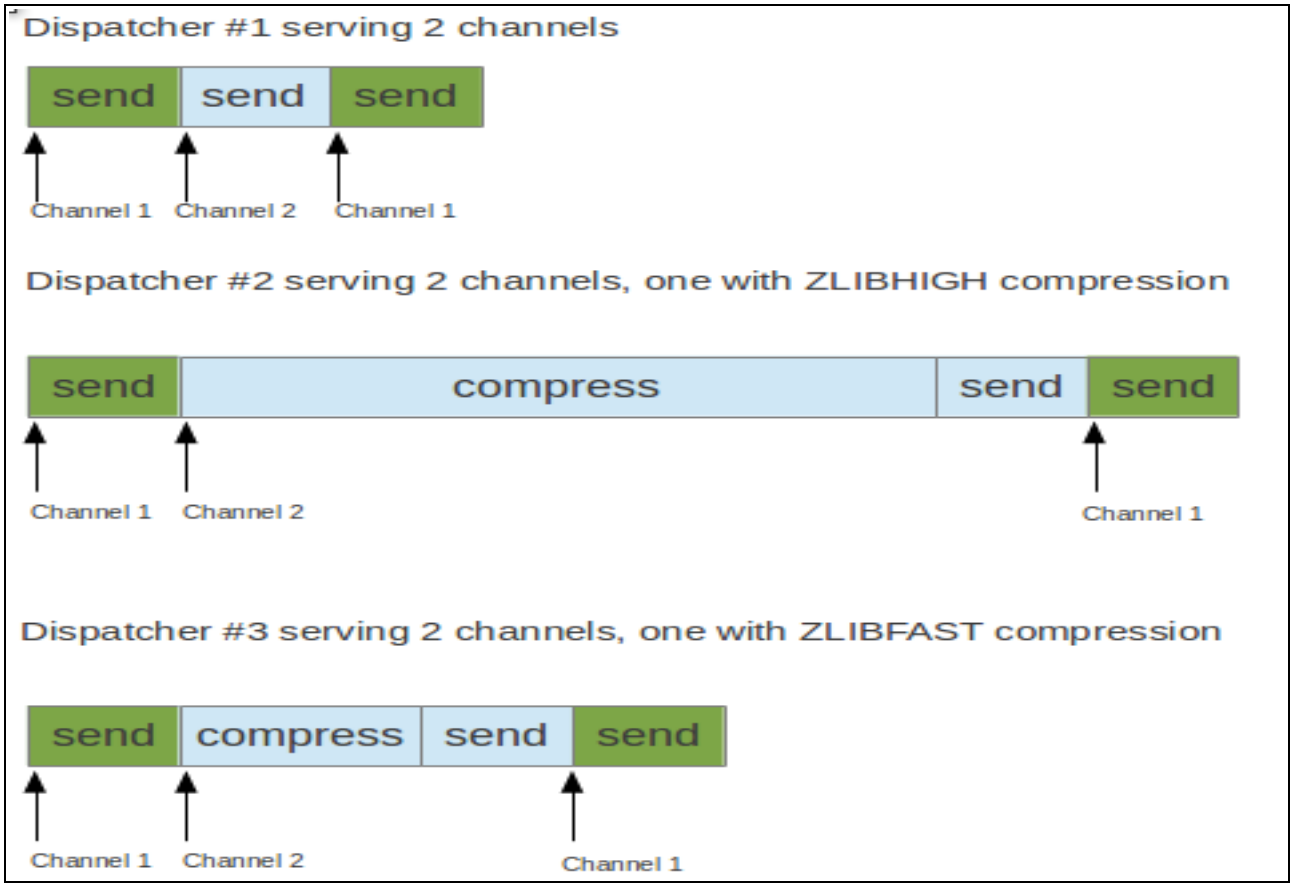
WebSphere MQ version 8.0 adds support for hardware compression via the channel compression option “ZLIBFAST”. Using zEDC hardware compression can reduce the message costs by 80% compared to compression performed in software.

Does message compression affect channel initiator tuning options?

Channel compression and decompression on the z/OS queue manager is performed by a dispatcher task.

The additional load of any compression or decompression attempts[9] therefore will place additional load on the dispatcher task performing that action. When that dispatcher is being used for multiple channels, this additional load can affect the performance of those other channels as they will have to wait for the compression / decompression to complete, e.g.

Diagram 2: Dispatcher processing multiple channels



9 Attempting compression, whether successful or not, will use additional CPU cycles.

This diagram represents 3 scenarios where each dispatcher task is supporting 2 outbound channels. *Note these times are not to scale.*

The size of the box labelled “compress” varies depending on the type of compression and has a direct impact on how quickly the dispatcher task can process the second green “send”.

This has several implications:

- A dispatcher task may become CPU constrained when supporting less channels when channel compression is involved. This can be determined using the channel initiator accounting and statistics data, in particular the dispatcher report, for example:

```
Task,Type,Requests,Busy %,     CPU used, CPU %,"avg CPU","avg ET"
   ,    ,        ,      ,      Seconds,      , uSeconds,uSeconds
   7,DISP,  146303,  98.5,    58.729109, 97.9,       401,     404
   8,DISP,  147030,  25.7,    15.514522, 25.9,       106,     105
```

  - The example shown is 1 outbound channel using ZLIBHIGH for compression (task 7) and 1 inbound channel using ZLIBHIGH for decompression (task 8)
  - In this example dispatcher task 7 is 98.5% busy for this interval and would be unlikely to support any more work.
  - Dispatcher 8 is 25.7% busy and could support more channels / workload.
- If a dispatcher is supporting multiple channels and one or more of those channels is using channel compression, the messages for those channels without channel compression may take longer to process. This may be identified by the “time on queue” and is effectively classed as “waiting for the dispatcher”. Altering the ratio of channels to dispatcher may resolve this delay.
- ZLIBFAST compression may be performed by zEDC and when this happens it appears to be synchronous but the operation is offloaded to the zEDC hardware from the CPU. *This means that the dispatcher task is blocked waiting for zEDC, and this will affect all channels using this dispatcher but also means that other tasks may use the CPU*. This can be determined from the channel initiator accounting and statistics data, in particular the dispatcher report where there is a difference between the CPU time and the elapsed time e.g.

```
Task,Type,Requests,Busy %,     CPU used, CPU %,"avg CPU","avg ET"
   ,    ,        ,      ,      Seconds,      , uSeconds,uSeconds
   7,DISP, 1443847,  71.6,    19.130398, 31.9,        13,      30
   8,DISP, 1431899,  20.4,    12.655084, 21.1,         9,       9
```

  - In this example, the difference between average elapsed time and average CPU time per request for dispatcher 7 is 17 microseconds which is the time spent compressing the message.
  - The time spent compressing (dispatcher 7) is significantly less than the earlier example for ZLIBHIGH (404 microseconds) as is the time spent decompressing the message (9

microseconds) compared to ZLIBHIGH which took 105 microseconds per segment.

Is there a minimum size message that can benefit from hardware compression?

When ZLIBFAST compression is specified, the hardware will only be used when the buffer exceeds certain threshold sizes. These thresholds can be specified in the IQPPRMxx parmlib member. The default settings in z/OS v2r1 are shown below:

	Parameter	***Default***	***Minimum size***
Compression	DEFMINREQSIZE	4 (KB)	4 KB
Decompression	INFMINREQSIZE	16 (KB)	4 KB

Should the message buffer to be passed into the compression or decompression routine be smaller than the appropriate threshold, the compression or decompression will be performed in the software layer. This can mean that compression is performed by zEDC and decompression is performed by the software layer.

It should also be noted that when preparing to send larger messages, the dispatcher task will process segments of the message. The maximum size of the segment will depend on whether SSL is enabled or not.

	Maximum segment size	***Segments in a 64KB message***
SSL Channel	16 KB	5
Non-SSL channel	32 KB	3

Note that the number of segments in a 64KB message is equal to:

```
(message size / maximum segment size) + 1
```

This is due to the implementation headers being added which take the actual of number of bytes sent to approximately 64.5KB.

This also means that each segment would be compressed and then sent, i.e. the number of send calls remains the same regardless of whether compression is enabled.

What about ZLIBHIGH?

There are 2 types of ZLIB compression supported by WebSphere MQ, namely ZLIBFAST and ZLIBHIGH.

- ZLIBFAST uses ZLIB encoding but prioritises on the speed of compression.
- ZLIBHIGH uses ZLIB encoding with the emphasis on achieving higher levels of compression. This more aggressive compression processing is performed in the software layer.

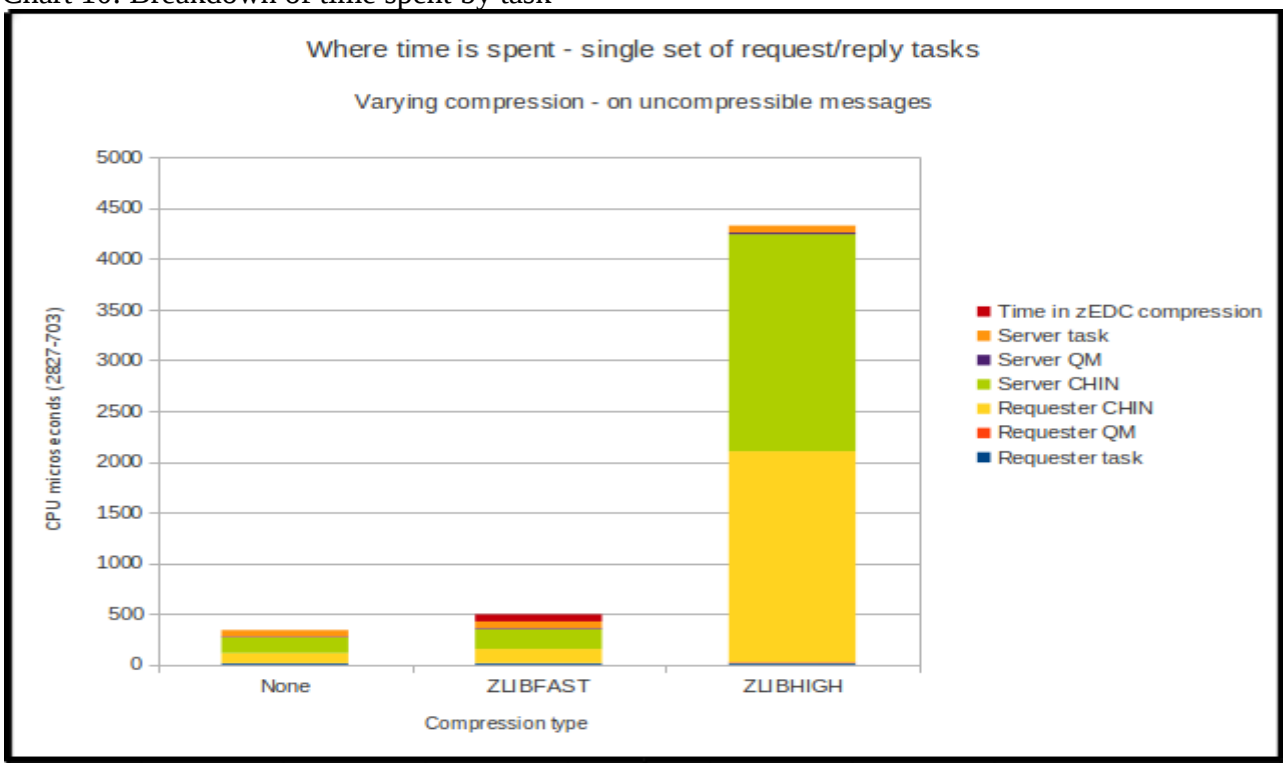
In certain circumstances however, the channel compression option “ZLIBHIGH” may use hardware for decompression.

Example: Comparing effect of compression types on non-compressible data.

Consider a request/reply workload between 2 remote queue managers where there is only a single message on the system. This scenario shows a good correlation between the round-trip time and the cost of processing the message on its journey.

- The scenario uses batch applications with no business logic.
- The requesting task puts the message out of syncpoint and waits for a specific reply message.
- The server task gets the next available message and puts a reply message within syncpoint.

Chart 10: Breakdown of time spent by task



Notes on chart:

- Message used is 64KB non-persistent and contains random data to be uncompressible.
- The time for a message to complete its processing increases by 50% when ZLIBFAST compression is attempted compared to using no compression. In this instance the ZLIBFAST processing did not achieve any compression, so no decompression was required.
- When the ZLIBHIGH channel compression is used, there is a significant increase (10 times) in the time for the message to complete its processing. This is entirely due to the time spent attempting to compress prior to sending, both at the requester and server side and, since ZLIBHIGH was able to compress the data by 4%, inflating the data using the software layer.
- Increasing the number of messages on the system whilst using the same pair of queues by adding more requesters and servers results in a greater difference between the CPU cost and the round-trip time. This is due primarily to the time spent waiting for the dispatcher to compress/decompress the message.
- The majority of transaction cost is these examples is in the channel initiator address spaces.

***Example:* Comparing effect of compression types on data that is compressible.**

A request / reply workload is run between 2 remote queue managers. There are multiple request tasks (each with 1 message at any time) and sufficient server tasks that the message is not delayed waiting for an application to process it.

In this example a transaction is defined as :

- a message generated and put by the requester,
- the message got by the server and a reply put (in-syncpoint)
- the reply message got by the originating requester.

Chart 11: Transaction rate using channel compression

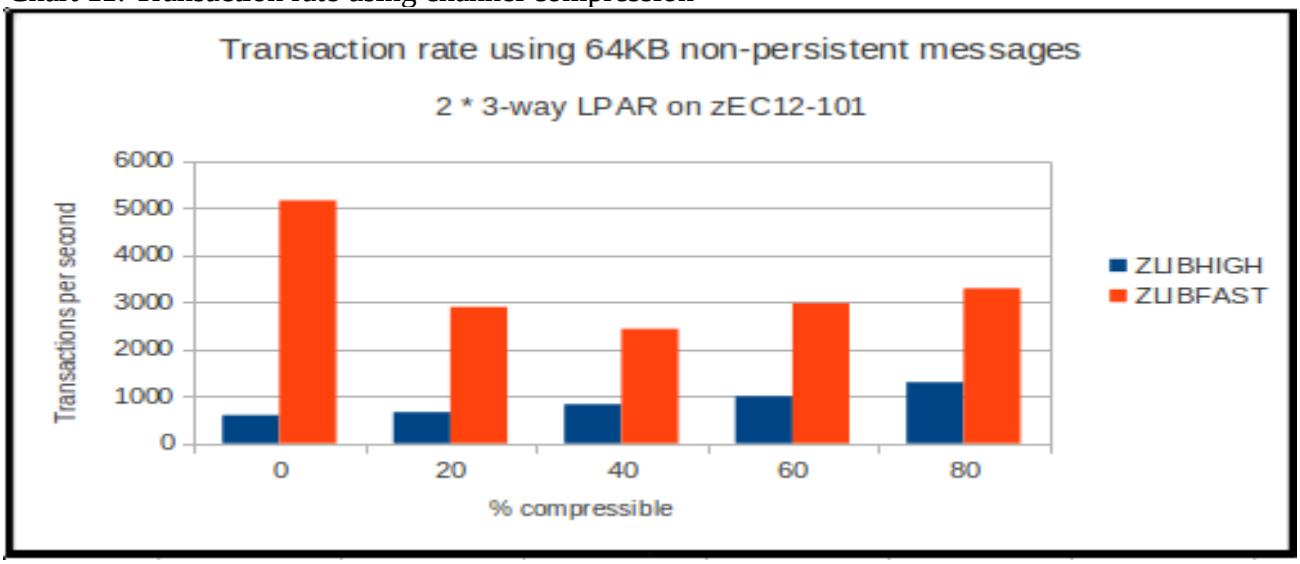
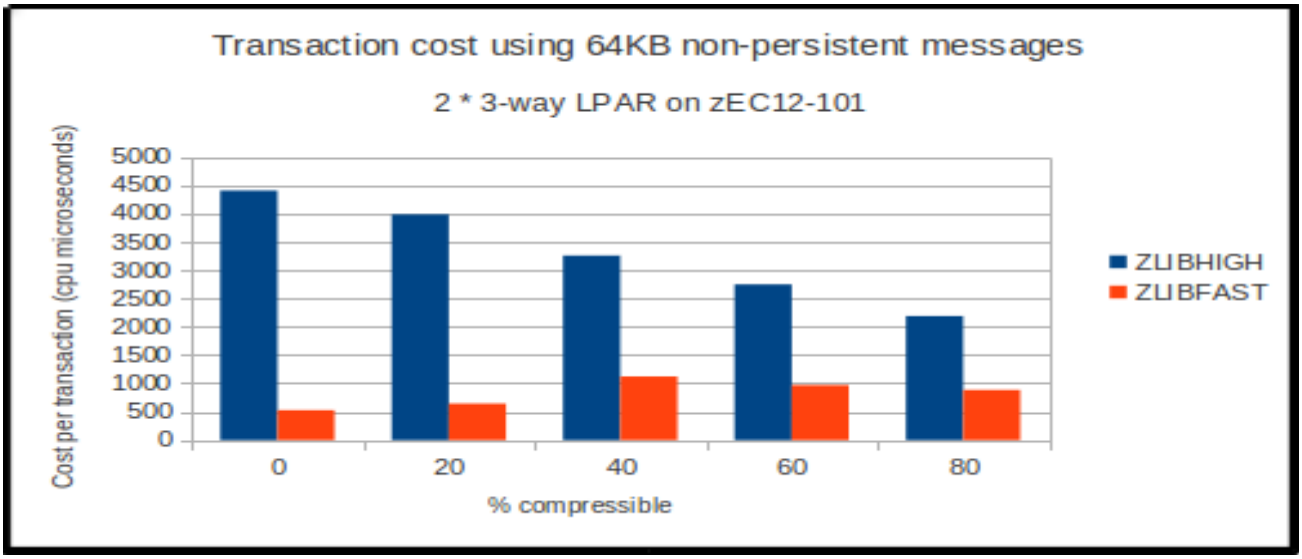


Chart 12: Transaction cost using channel compression



Notes on charts:

- ZLIBFAST shows a significant increase in transaction rate for all levels of compression,

ranging from 2.5 times to 8.5 times that of ZLIBHIGH.

- The transaction rate at 40% compressible is lower for ZLIBFAST than at 20% compressible as that is the point at which the decompression first takes place in the software layer. After this point decompression takes place only in the software layer. This corresponds with the increase in transaction cost.

Example: *Comparing effect of altering compression thresholds*

This is an extension to the previous example as it now includes measurements where the INFMINREQSIZE parameter has been altered to specify the minimum value of 4KB.

Chart 13: Transaction rate using channel compression whilst varying threshold

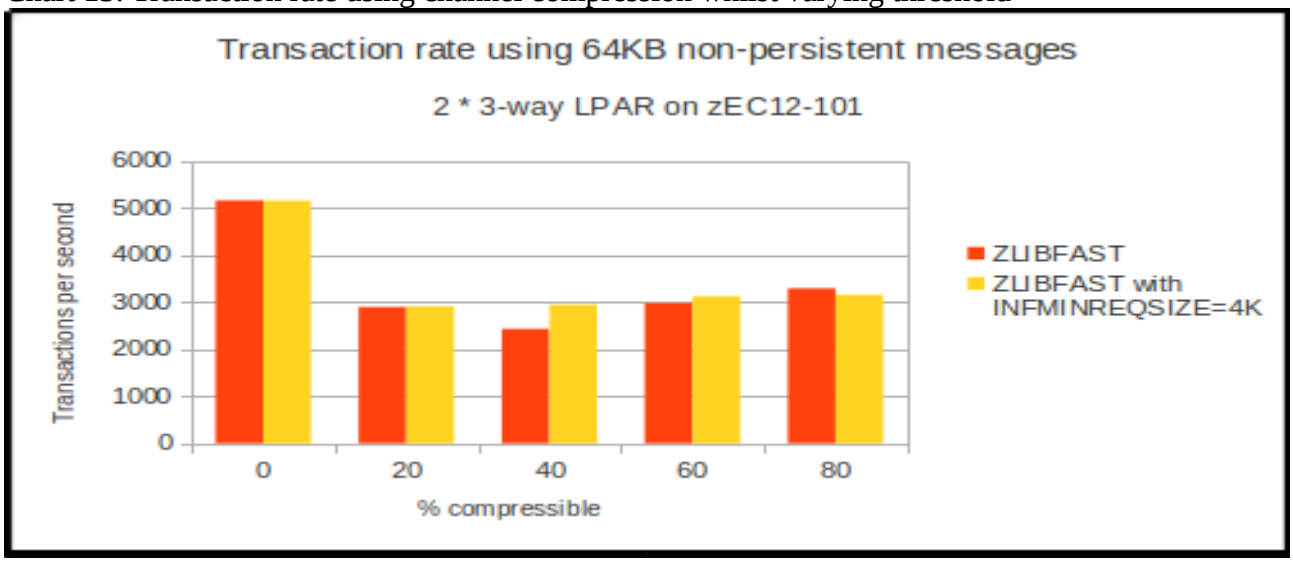
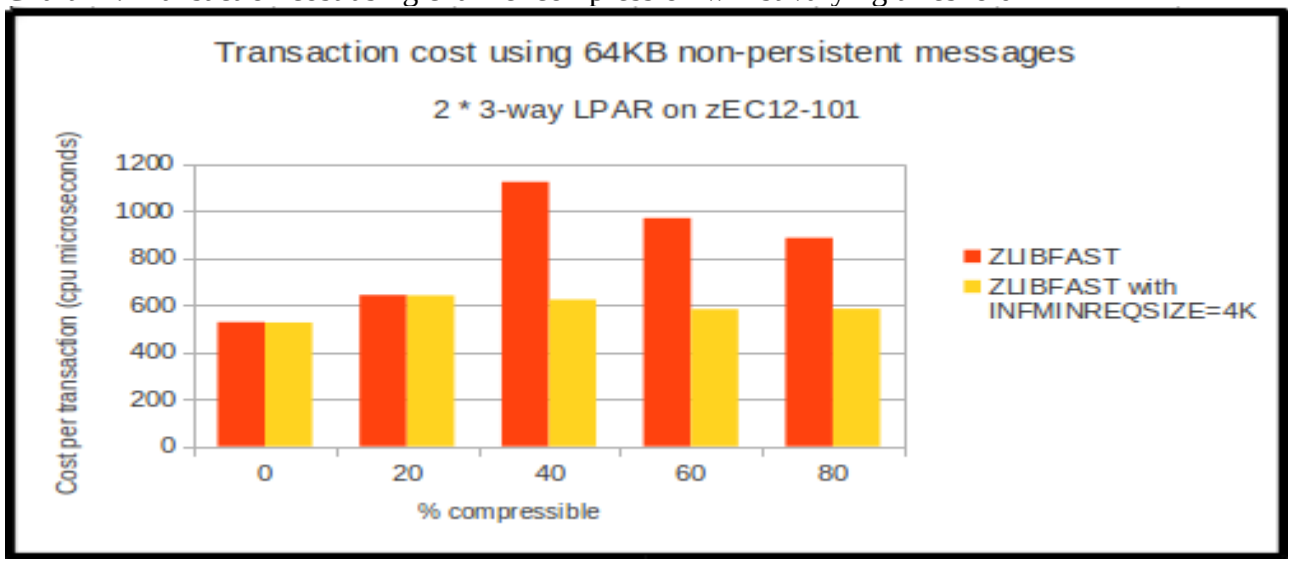


Chart 14: Transaction cost using channel compression whilst varying threshold



Notes on chart 13 and 14:

- Lowering the inflation threshold helped the 40% compressible message workload by increasing the transaction rate by 20%.
- Performing the inflation in the hardware also reduced transaction cost by between 30 and 45% for those messages that are more compressible.
- As these messages are being compressed for non-SSL channels, the segment size is 32KB.
  - Even 32KB segments that are 80% compressible can be inflated using hardware.
  - Once the compression rate exceeds 87%, the decompression will be performed in software.

Example: Using hardware compression on SSL enabled channels

The CPU costs incurred compressing a message are likely to outweigh the CPU savings in the communications layer when moving a smaller compressed message.

However there are use cases where hardware compression can result in a reduction in the overall transaction cost compared to the uncompressed message, for example when using SSL encrypted channels.

When message compression and SSL are enabled for a channel, the dispatcher will compress the segment before the data is encrypted.

Performance report MP16 “Capacity planning and tuning guide for WebSphere MQ for z/OS” discusses the cost of SSL secret key negotiation as well as the relative cost of encrypting the data.

Typically the cost of a secret key negotiation on a 2827-703 is 10-12 CPU microseconds.
If the secret key is set to renegotiate at 1MB boundaries and the message size is 64KB:

- we would expect to renegotiate every 16 messages
- and we can predict a secret key negotiation cost of approximately 600-750 microseconds per message.

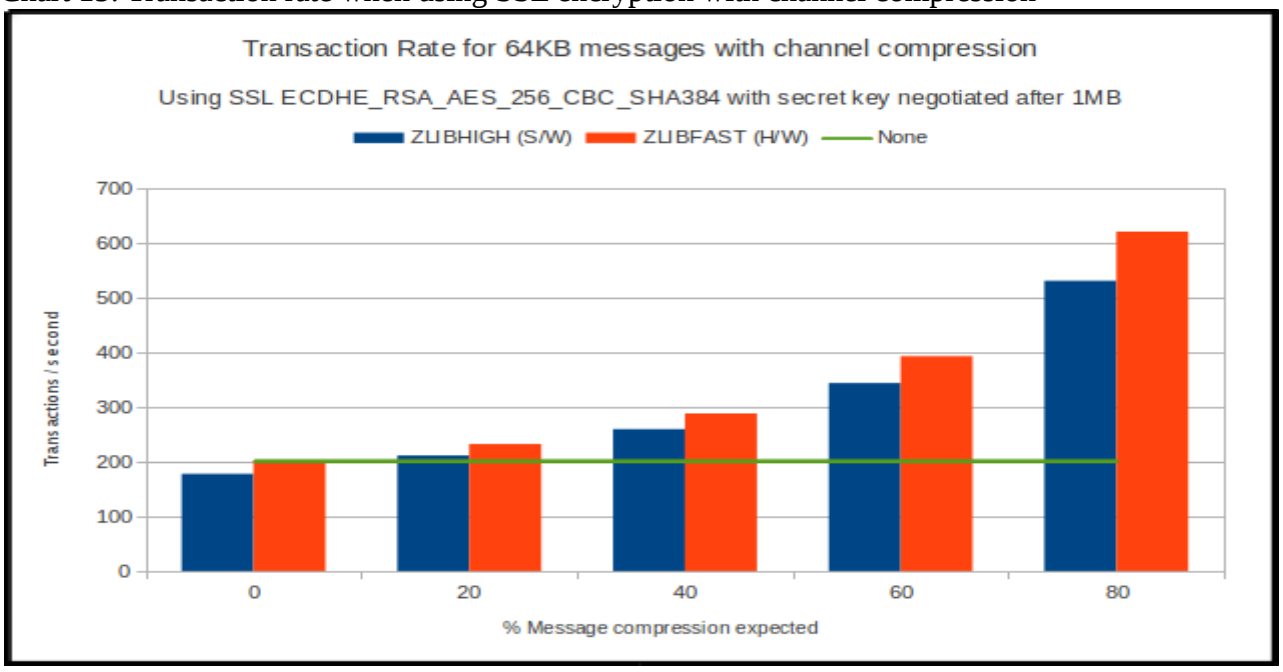
By compressing the message it is possible to increase the number of messages flowed between renegotiations, e.g. if a 64KB message is 40% compressible, it would be be compressed to approximately 38.4KB which means:

- the secret key renegotiation takes place every 27 messages
- the cost of the negotiation would be 370-444 microseconds per message, a saving of 250-300 microseconds per message.

The following 2 charts show the achieved transaction rate and cost when using a request/reply model to send a 64KB non-persistent message between 2 remote queue managers.

- The secret key is being renegotiated every 1MB.
- The degree of message compressibility increases, resulting in more messages flowing per secret key renegotiation.
- Both charts show the achieved rate and cost when no compression is enabled.

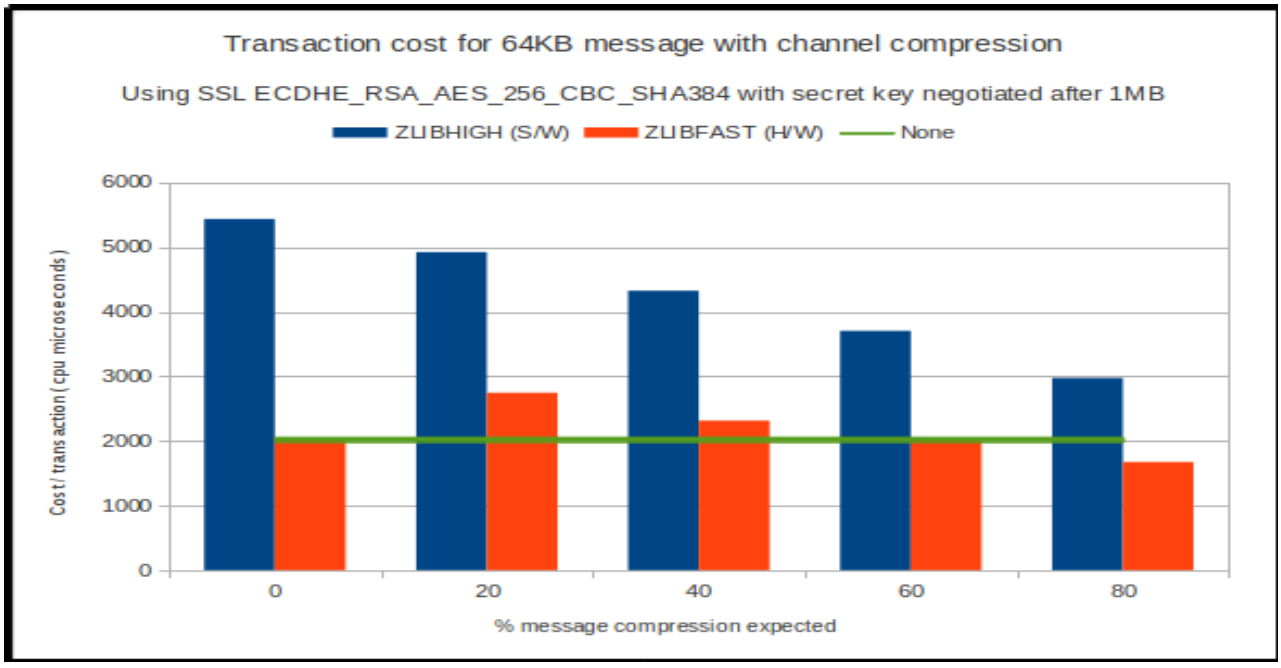
Chart 15: Transaction rate when using SSL encryption with channel compression



Notes on chart 15:

- The systems under test are not CPU constrained
- Transaction rate when using ZLIBFAST even for uncompressible messages matches the rate when no compression is attempted.
- Once the messages are compressible, the rate achieved exceeds the rate when no compression is attempted and exceeds it by over 3 times at its peak.
- Reducing the frequency of the secret key renegotiations would diminish the benefits of compression.

Chart 16: Transaction cost when using SSL encryption with channel compression



Notes on chart 16:

- The cost of the transaction when compressing the messages using software is significantly higher than either using hardware compression or no compression.
- The cost of hardware compression matches the cost of no compression when the message is approximately 50% compressible.
- The decompression of the messages is always performed in the software layer. By altering the INFMINREQSIZE threshold to 4KB, further reductions in cost could be made for messages up to 75% compressible (as the SSL segment is only 16KB).
- The costs include the resources on both LPARs, therefore they include the cost of compressing and decompressing the message twice.

Using the channel initiator accounting and statistics data we can break down the costs by task.

The 40% compression level has been used as this is where the costs of compressing the message prior to SSL encryption are approximately equal to the non-compression case.

Outbound Channel (compressing)

	Dispatcher cost	***SSL Task***	***Total***
SSL only	75	232	307
SSL with hardware compression (achieving 40% compression)	121	195	316
SSL with software compression (achieving 40% compression)	1140	180	1320

Inbound Channel (decompressing)

	Dispatcher cost	***SSL Task***	***Total***
SSL only	50	210	260
SSL with hardware decompression (where the message was 40% compressible)	350	150	500
SSL with software decompression (where the message was 40% compressed)	350	150	500

Notes:

- These costs exclude the Cryptographic co-processor costs for SSL key renegotiation and the zEDC processor used for hardware compression.
- Costs are in CPU microseconds.
- All decompression was performed in the software layer as the default inflation threshold was used (16KB) and SSL segments are a maximum size of 16KB.

Appendix A – Regression

When performance monitoring WebSphere MQ version 8 for z/OS, a number of different categories of tests are run, which include:

- Private Queue
- Shared Queue
- Moving messages using MCA Channels
  - SSL
  - Channel compression (ZLIBFAST / ZLIBHIGH)
- Moving messages using Clustered Channels
- Client
- Bridges and Adaptors
- Trace

These tests are run against versions 7.0.1 (v701), 7.1 (v710) and 8.0 (v800) and the comparison of the results is shown in subsequent pages.

The statement of regression is based upon these results.

All measurements were run on a performance sysplex of a zEnterprise EC12 (2827-7A1) which was configured as described in Appendix B.

Given the complexity of the z/OS environment even in our controlled performance environment, a tolerance factor of +/-6% is regarded as acceptable.

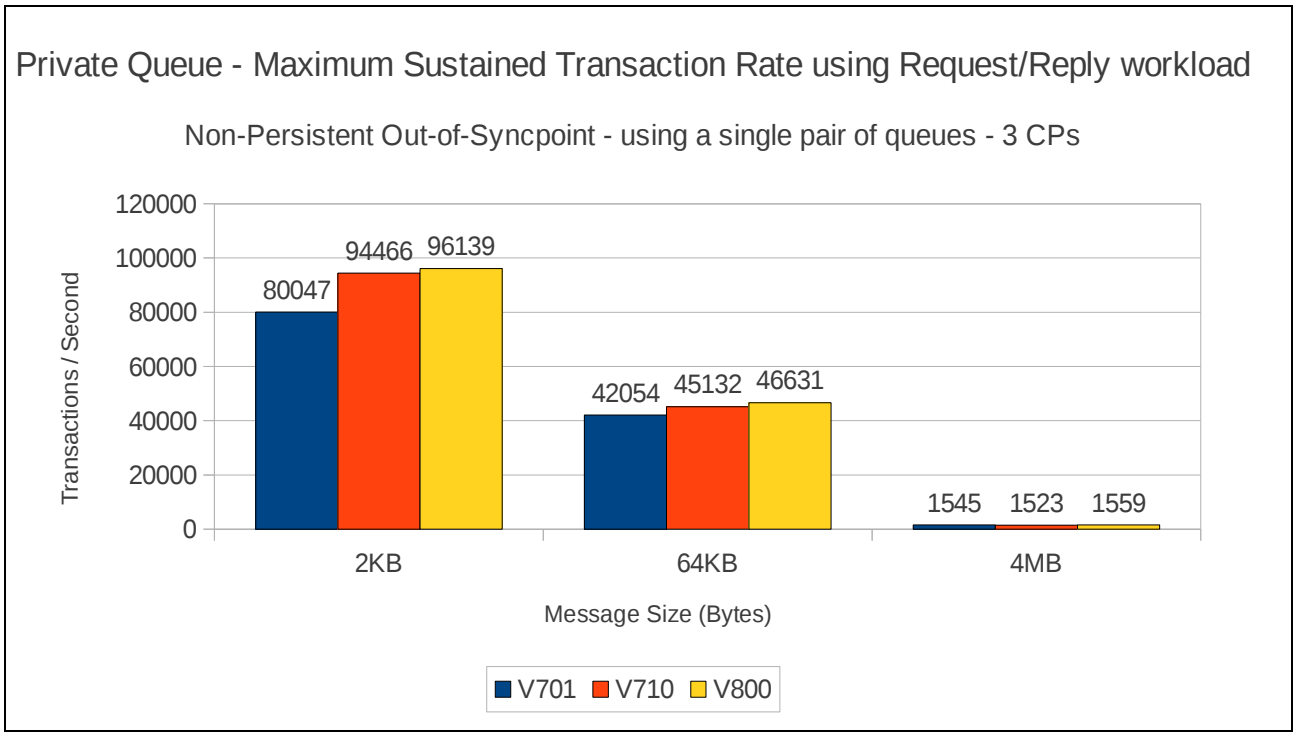
Regression – private queue

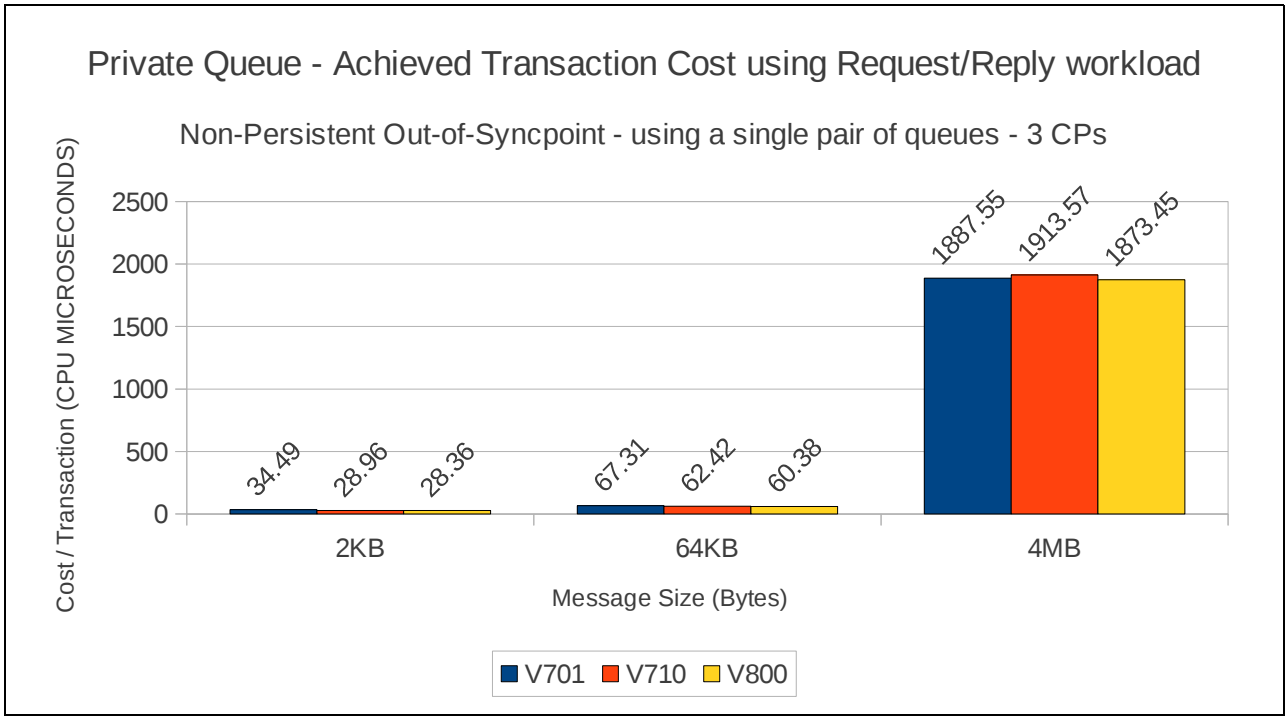
Non-Persistent Out-of-Syncpoint workload

Maximum Throughput on single pair of request/reply queues

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have gotten the message, they put another message to the request queue. The messages are put and got out of syncpoint.

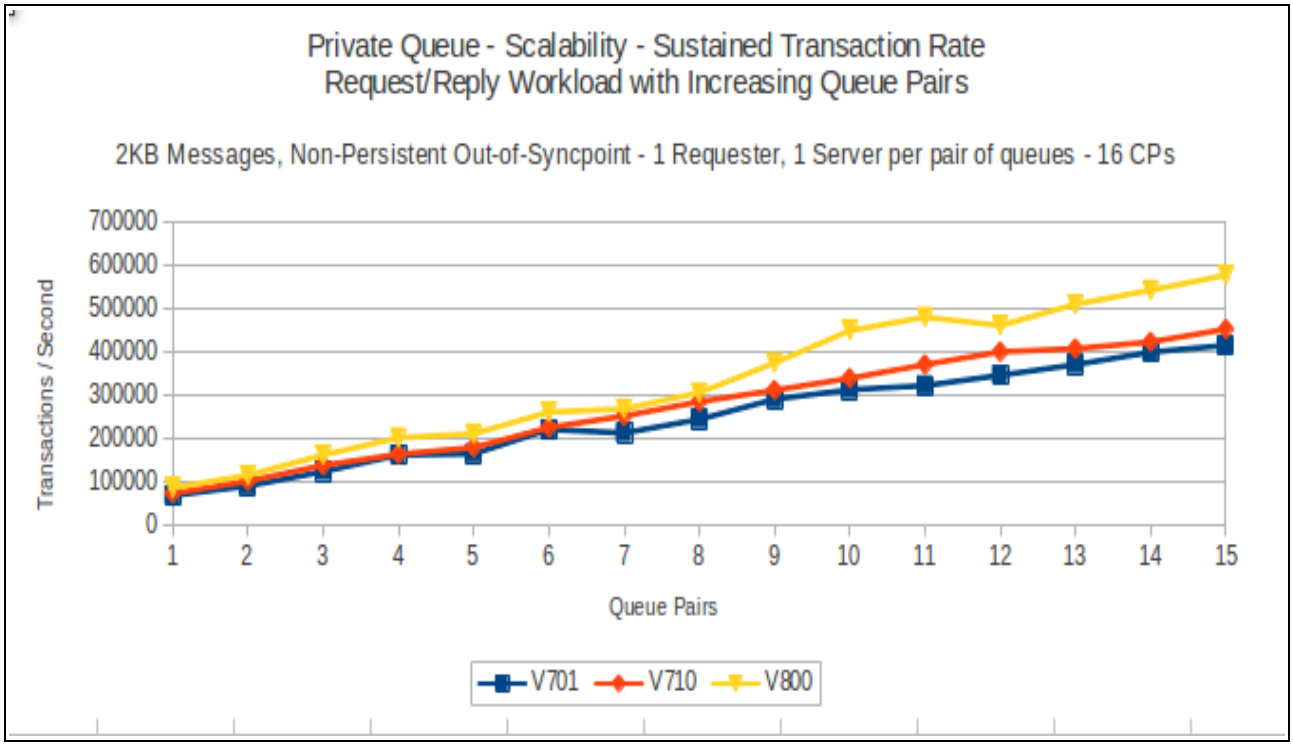
There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into an MQGET-with-wait. The messages are got and put out of syncpoint.

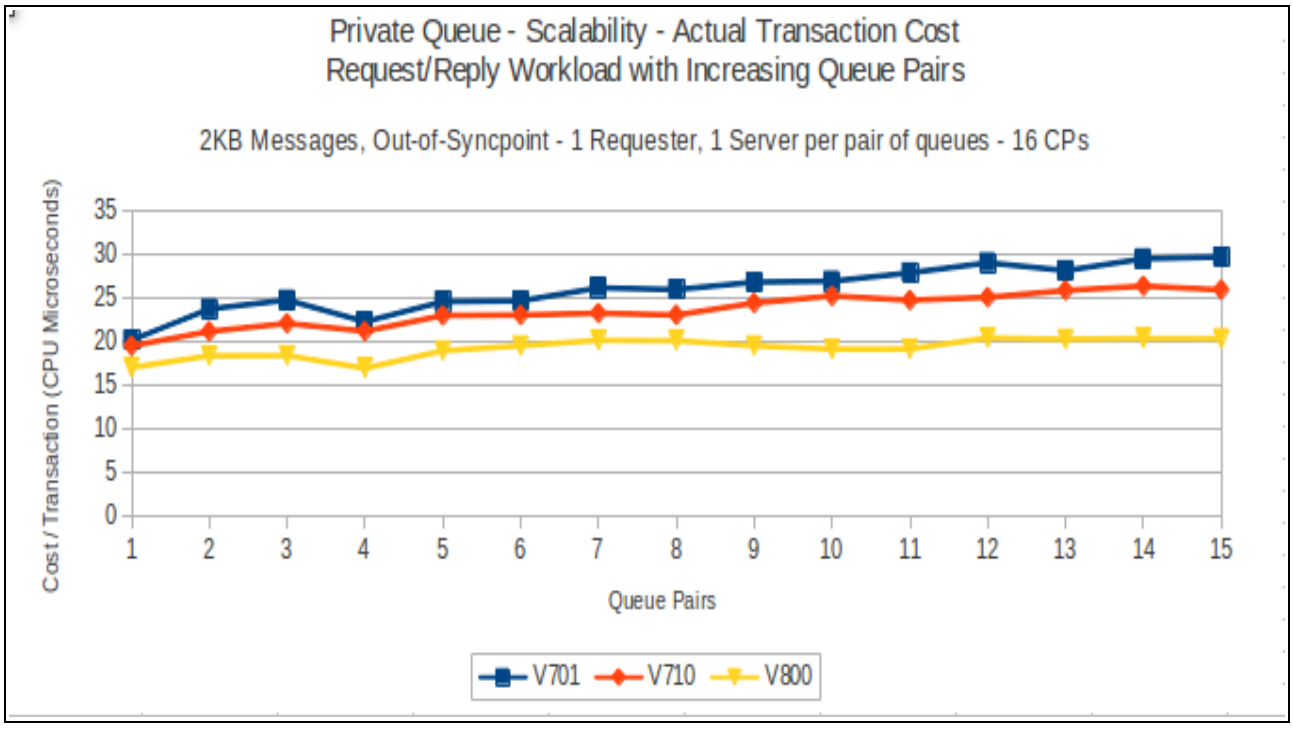




Scalability of request/reply model across multiple queues

The queue manager is configured with 16 pagesets – 0 through 15 and a corresponding number of buffer pools.
On each of pagesets 1 to 15, a pair of request and reply queues are defined. The test starts up 1 requester and 1 server task accessing the queues on pageset 1 and runs a request/reply workload. At the end, the test starts a second pair of requester and server tasks which access the queues on page set 2 and so on until there are 15 requester and 15 server tasks using queues on all 15 pagesets with application queues defined.
The requester and the server tasks specify NO_SYNCPOINT for all messages.
The measurements are run on a single LPAR with 16 dedicated processors online on the zEnterprise EC12 (2827) used for testing.





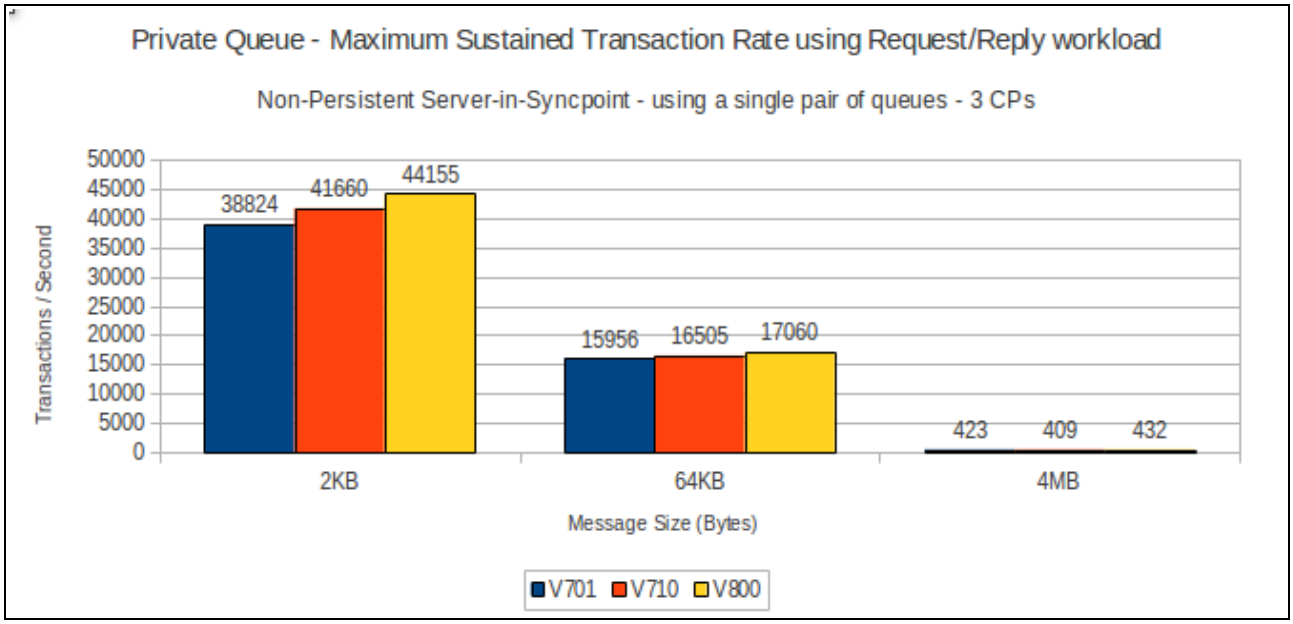
The preceding 2 charts show that a single queue manager is able to drive in excess of 575,000 transactions per second – or 1,100,000 non-persistent messages per second on a 16-way LPAR.

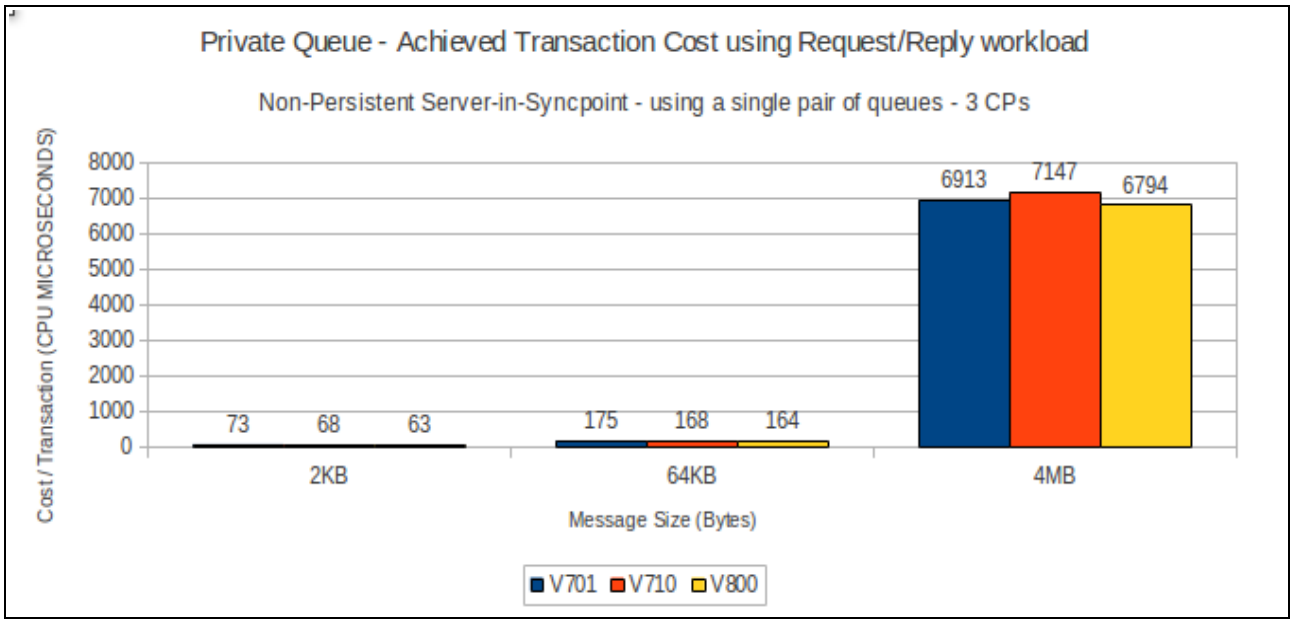
Non-persistent server in-syncpoint workload

Maximum throughput on single pair of request/reply queues

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have got the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that use MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into an MQGET-with-wait. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.





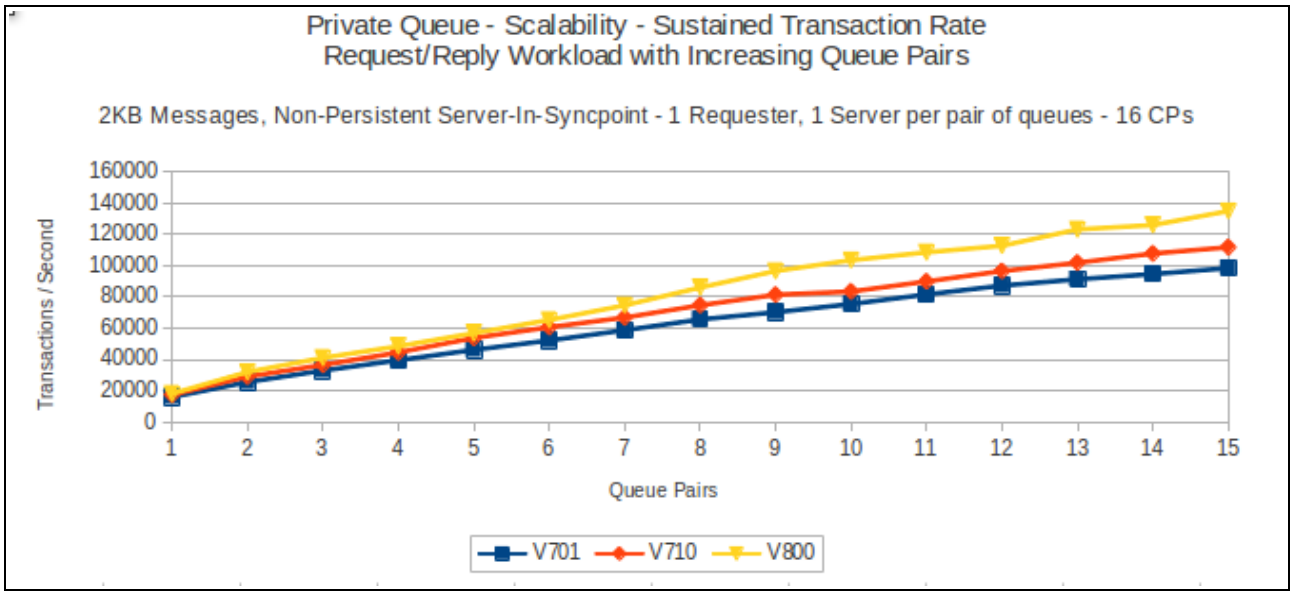
Scalability of Request/Reply model across multiple queues

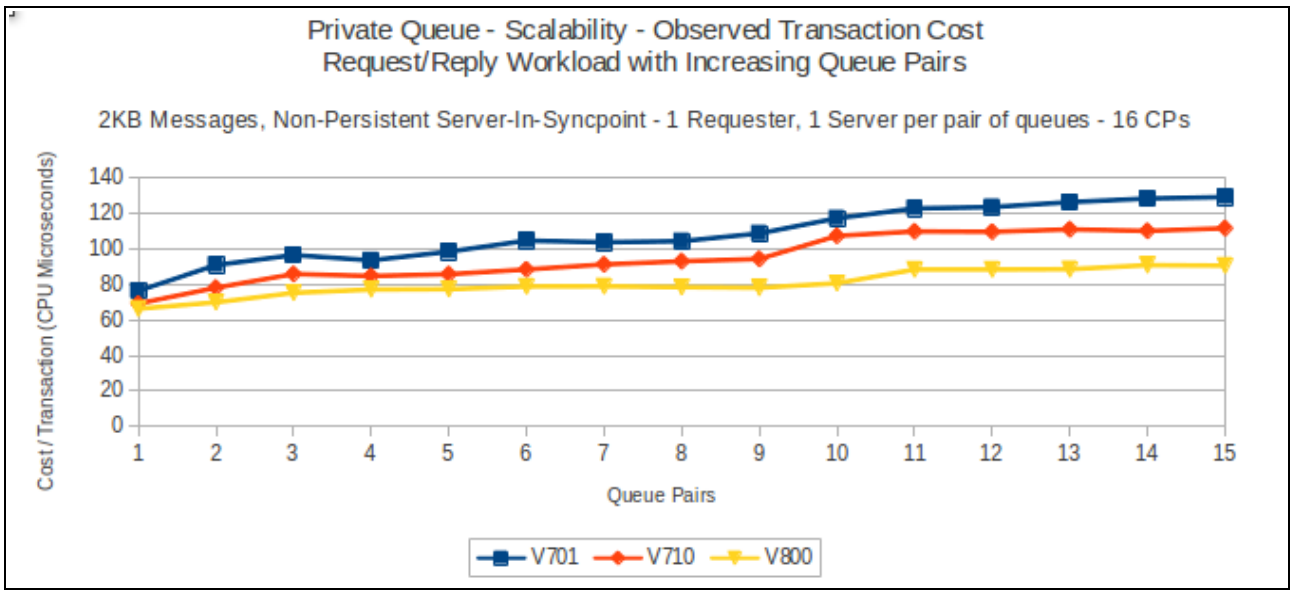
The queue manager is configured with 16 pagesets – 0 through 15 and a corresponding number of buffer pools.

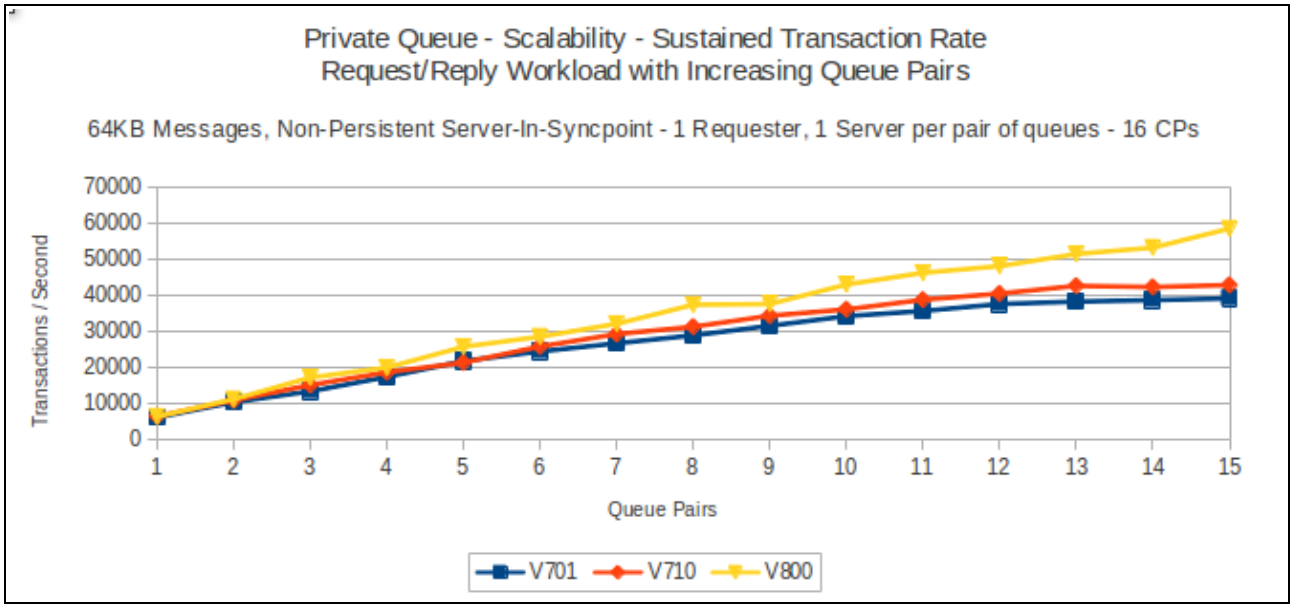
On each of pagesets 1 to 15, a pair of request and reply queues are defined. The test starts up 1 requester and 1 server task accessing the queues on pageset 1 and runs a request/reply workload. At the end, the test starts a second pair of requester and server tasks which access the queues on page set 2 and so on until there are 15 requester and 15 server tasks using queues on all 15 pagesets with application queues defined.

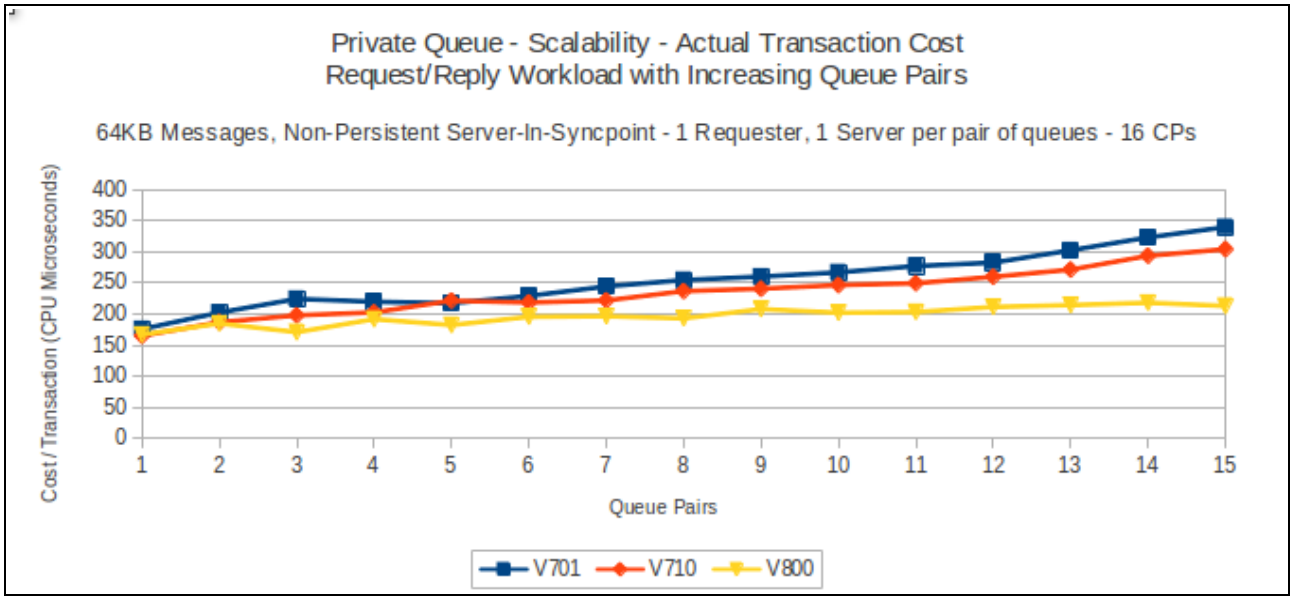
The requester tasks specify NO_SYNCPOINT for all messages and the server tasks get and put messages within syncpoint.

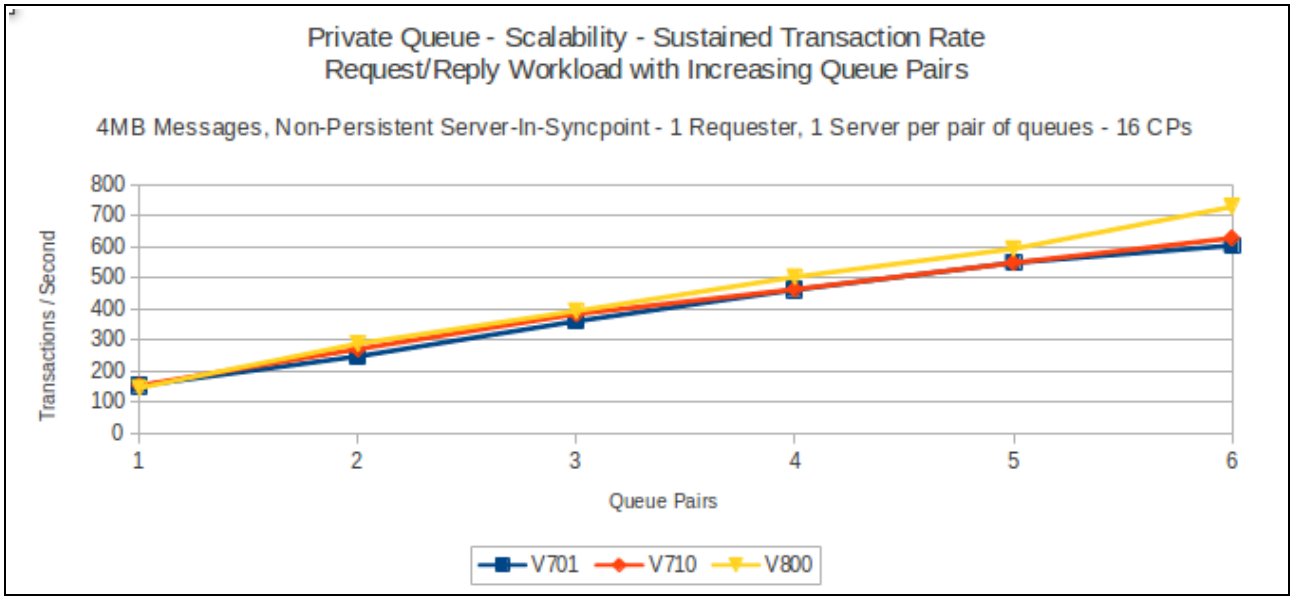
The measurements are run on a single LPAR with 16 dedicated processors online on the zEnterprise EC12 (2827) used for testing.

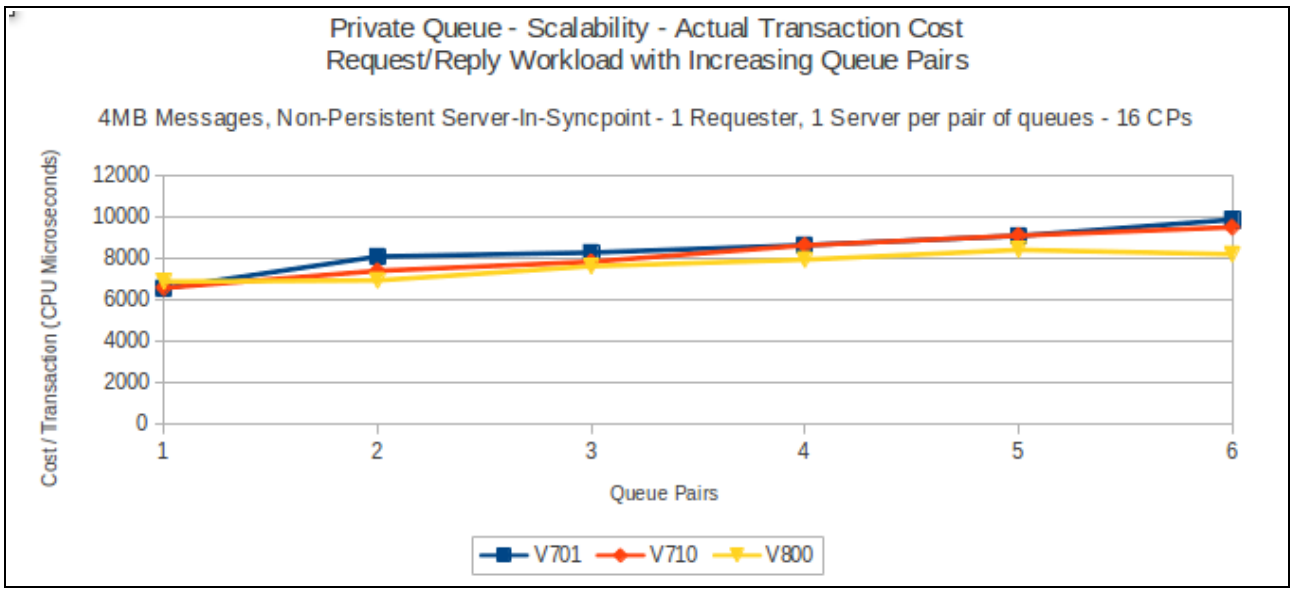





Private Queue - Scalability - Sustained Transaction Rate
Request/Reply Workload with Increasing Queue Pairs
64KB Messages, Non-Persistent Server-In-Syncpoint - 1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
70000
60000
50000
40000
30000
20000
10000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Queue Pairs
V701
V710
V800


Private Queue - Scalability - Actual Transaction Cost
Request/Reply Workload with Increasing Queue Pairs
64KB Messages, Non-Persistent Server-In-Syncpoint - 1 Requester, 1 Server per pair of queues - 16 CPs
Cost / Transaction (CPU Microseconds)
400
350
300
250
200
150
100
50
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Queue Pairs
V701
V710
V800


Private Queue - Scalability - Sustained Transaction Rate
Request/Reply Workload with Increasing Queue Pairs
4MB Messages, Non-Persistent Server-In-Syncpoint - 1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
800
700
600
500
400
300
200
100
0
1
2
3
4
5
6
Queue Pairs
V701
V710
V800

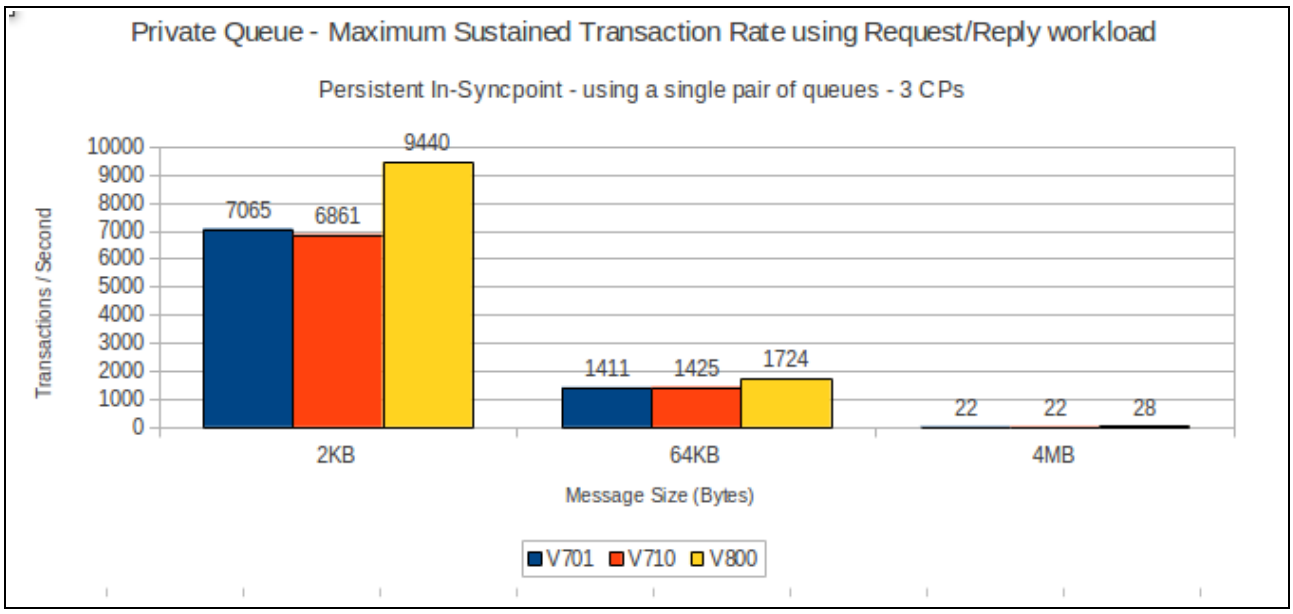

Private Queue - Scalability - Actual Transaction Cost
Request/Reply Workload with Increasing Queue Pairs
4MB Messages, Non-Persistent Server-In-Syncpoint - 1 Requester, 1 Server per pair of queues - 16 CPs
Cost / Transaction (CPU Microseconds)
12000
10000
8000
6000
4000
2000
0
1
2
3
4
5
6
Queue Pairs
V701
V710
V800

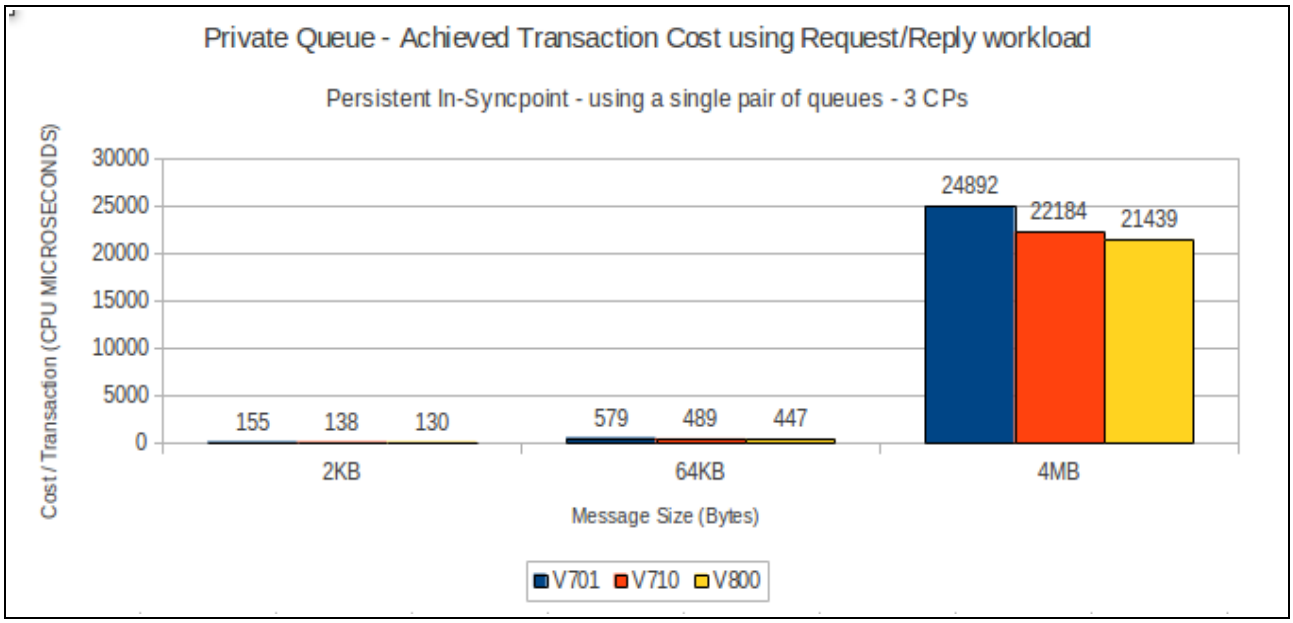
Persistent in-syncpoint workload

Maximum throughput on single pair of request/reply queues

The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have got the message, they put another message to the request queue. The messages are put in-syncpoint and got in-syncpoint.

There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into an MQGET-with-wait. The messages are got and put in syncpoint with 1 MQGET and 1 MQPUT per commit.

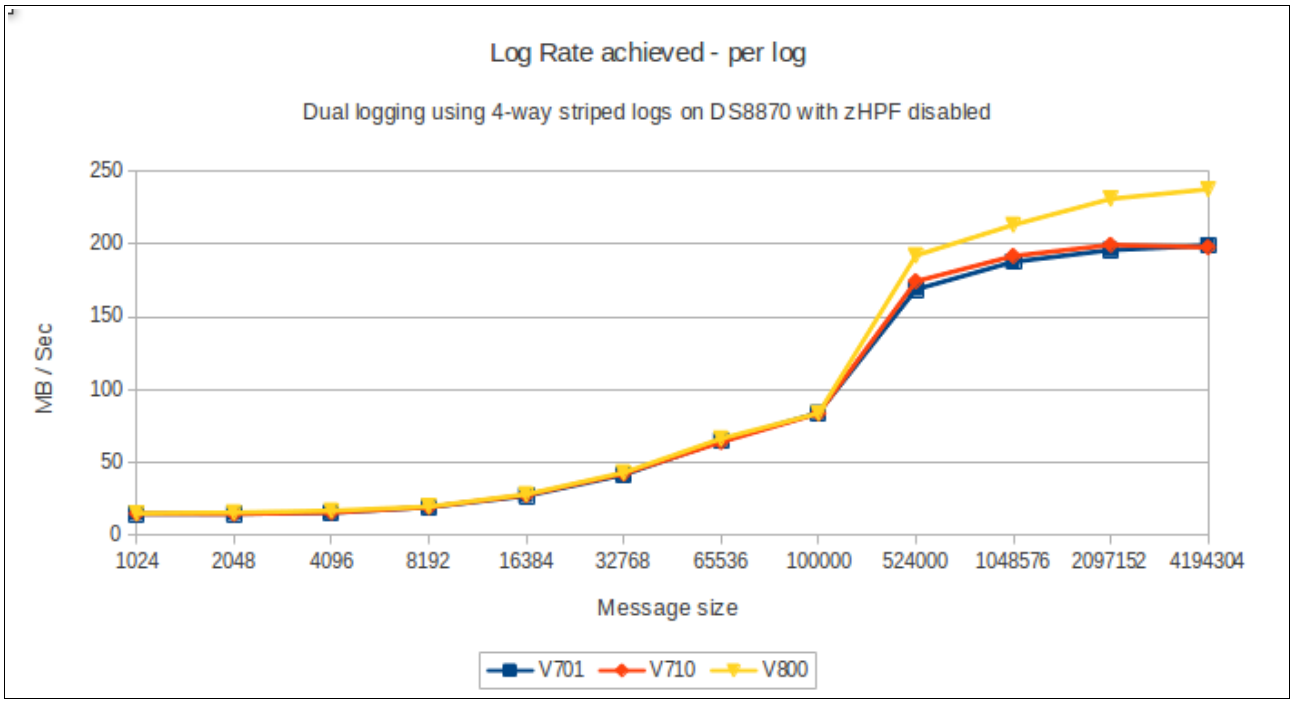




Upper bounds of persistent logging rate

This test uses 3 batch tasks that each put and get messages from a common queue. One of the tasks only uses a 1KB persistent message. The remaining 2 tasks vary the size of their message from 1KB to 4MB.

The following chart shows the achieved log rate on a 3-way LPAR on a zEC12.



The log rate achieved in version 8.0 begins to exceed that of previous releases with messages of 511KB or larger and in our measurements is able to exceed the peak logging rate of version 7.1 by 20%.

CICS Workload

When a CICS transaction makes a call using an MQ API, the processing performed at end of task can have a significant effect on the transaction cost. This cost is most noticeable when adding the first MQ call, e.g. MQPUT1 to the application.

In version 8.0 the processing performed by the queue manager at end of CICS task has been amended for scalability purposes, which reduces the impact of adding the first MQ call.

In previous releases on high n-way machines with high volume CICS transactions, the end of task processing would elongate as the workload increased. This was due to more storage being allocated for CICS tasks, because they were taking longer, and more time to scan that storage. In the worst case scenarios, the queue manager storage could rapidly be used, resulting in a queue manager failure. One solution was to limit the number of CICS transactions running at any point by configuring MAXOPENTCBS in the DFHSIT. Note, in CICS Transaction Server version 5.1, the MAXOPENTCBS option was made obsolete, being superceded by a value based on the MXT parameter[10].

In WebSphere MQ version 8.0, the storage is being more efficiently re-used which reduces the CPU cost per transaction and increases the throughput capacity.

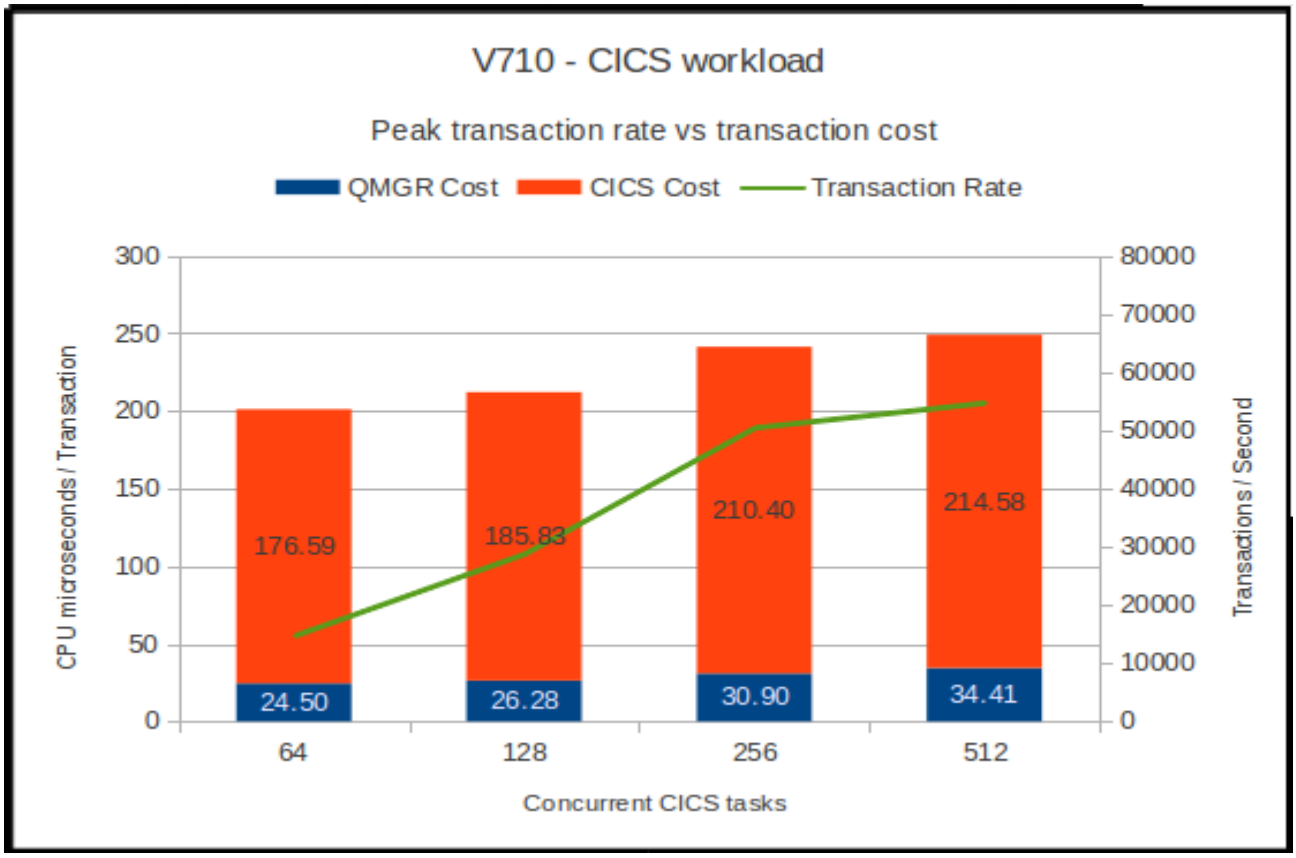
The following charts use a set of CICS transactions that open queues, put a 2KB non-persistent messages, get a 2KB non-persistent message, close queues and initiate a new CICS transaction. Running multiple sets of these transactions simulates a number of concurrent transactions across multiple CICS regions.

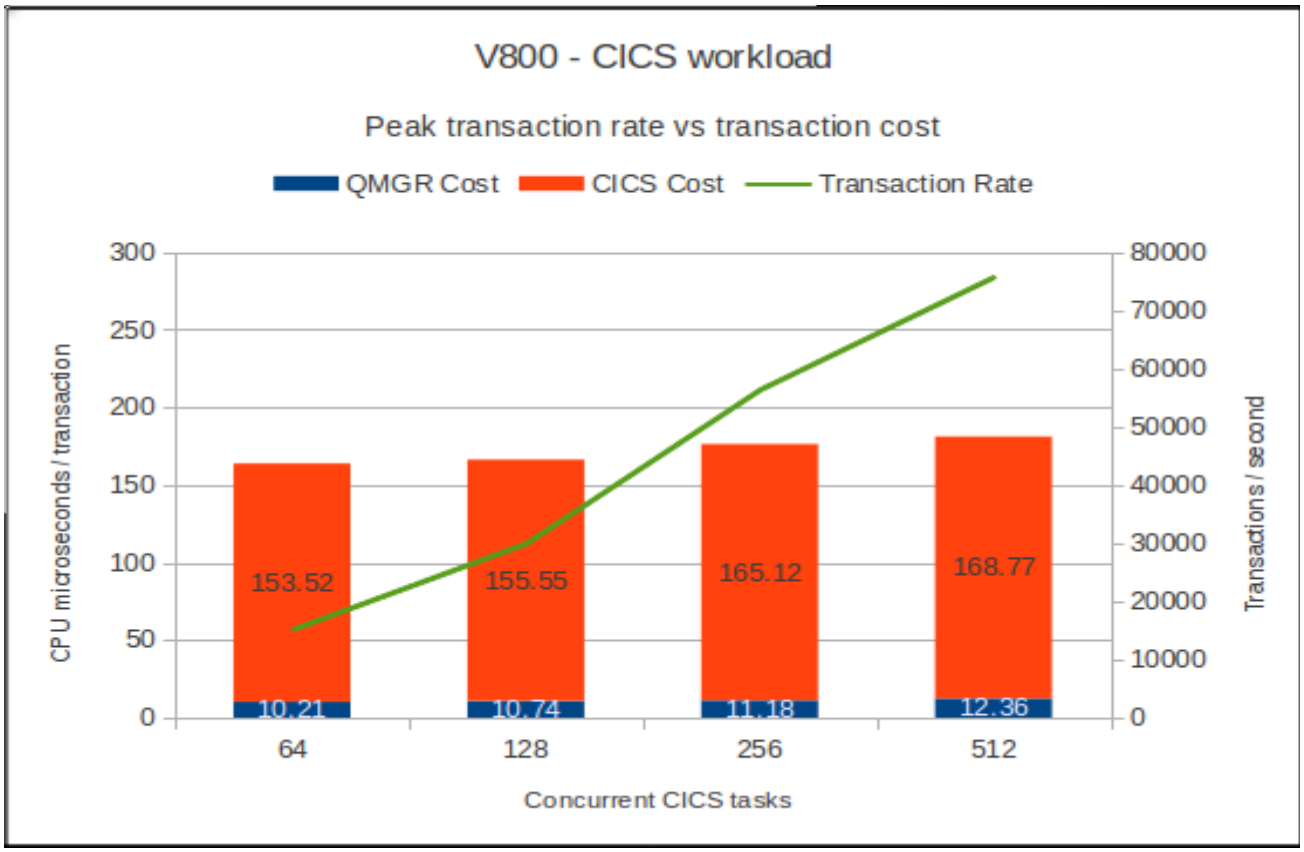
In each case the system is running with 16 dedicated processors and becomes CPU constrained with 512 concurrent CICS tasks.

Notes on charts:

- The scale used on each chart is consistent.
- For version 7.1, the queue manager cost associated with the CICS workload is approximately 25 CPU microseconds and this rises as the workload increases.
- For version 8.0, the queue manager cost associated with the CICS workload is 10 CPU microseconds and rises at a similar rate (percentage) as the workload increases.
- There is also an associated saving in the transaction cost, which means that in a CPU constrained system, the upper bound on transaction rate is raised by nearly 40% in these measurements.

10 The value for MAXOPENTCBS in CTS 5.1 onwards is calculated by CICS and is equal to (MXT * 2) +32.


V710 - CICS workload
Peak transaction rate vs transaction cost
QMGR Cost
CICS Cost
Transaction Rate
CPU microseconds / Transaction
Transactions / Second
300
250
200
150
100
50
0
80000
70000
60000
50000
40000
30000
20000
10000
0
176.59
185.83
210.40
214.58
24.50
26.28
30.90
34.41
64
128
256
512
Concurrent CICS tasks


V800 - CICS workload
Peak transaction rate vs transaction cost
QMGR Cost
CICS Cost
Transaction Rate
CPU microseconds / transaction
Transactions / second
300
250
200
150
100
50
0
80000
70000
60000
50000
40000
30000
20000
10000
0
153.52
155.55
165.12
168.77
10.21
10.74
11.18
12.36
64
128
256
512
Concurrent CICS tasks

Regression – shared queue - CFLEVEL(5)

Version 7.1.0 introduced CFLEVEL(5) with Shared Message Data Sets (SMDS) as an alternative DB2 as a way to store large shared messages. This resulted in significant performance benefits with both larger messages (greater than 63KB) as well as small messages when the Coupling Facility approached its capacity by using tiered thresholds for offloading data. For more details on the performance of CFLEVEL(5), please refer to performance report MP1H.

CFLEVEL(5) with SMDS will be used where available for the shared queue measurements.

Non-persistent out-of-syncpoint workload

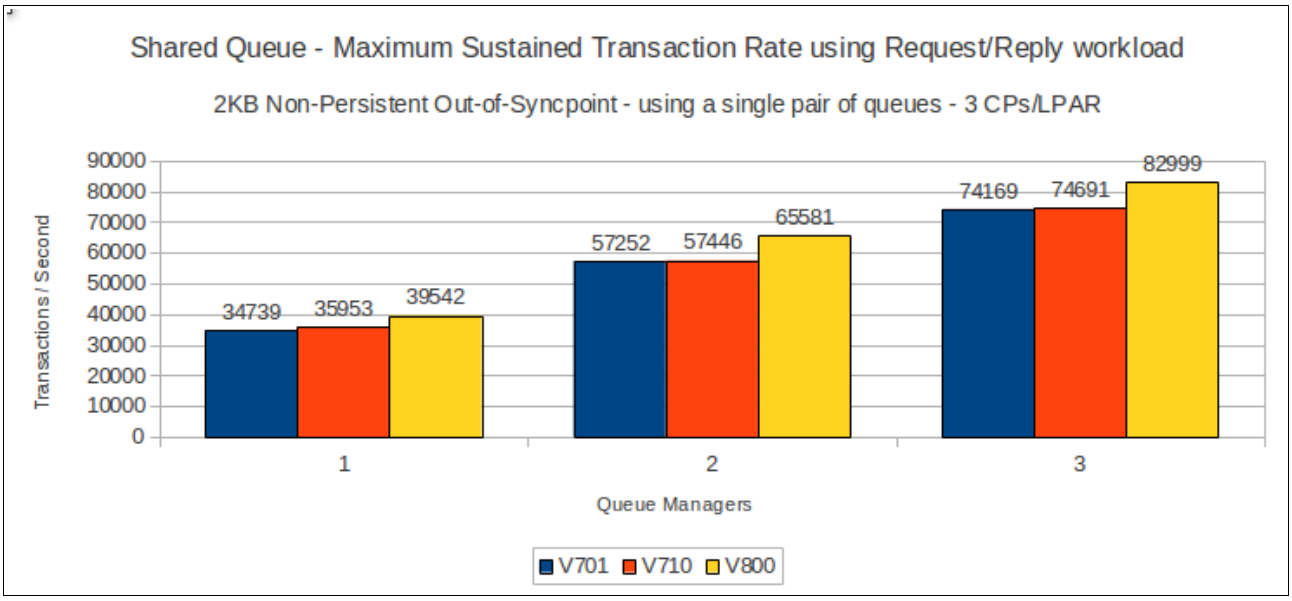
Maximum throughput on single pair of request/reply queues

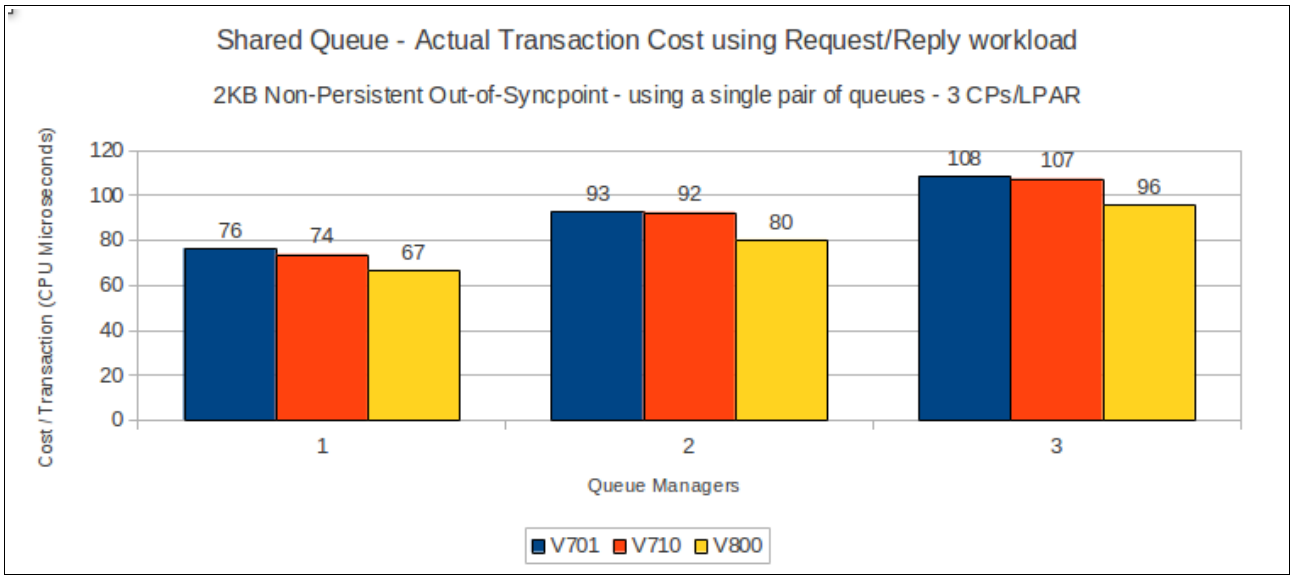
The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have got the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that use MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into an MQGET-with-wait. The messages are got and put out of syncpoint.

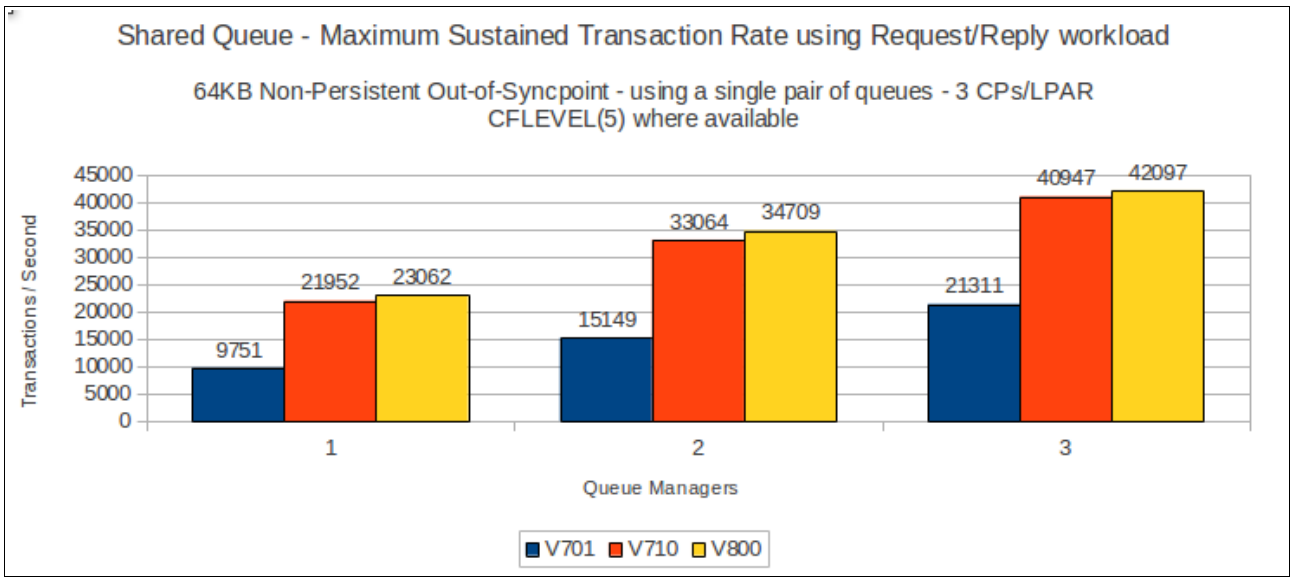
An increasing number of queue managers are allocated to process the workload. Each queue manager has 5 requester and 4 server tasks.

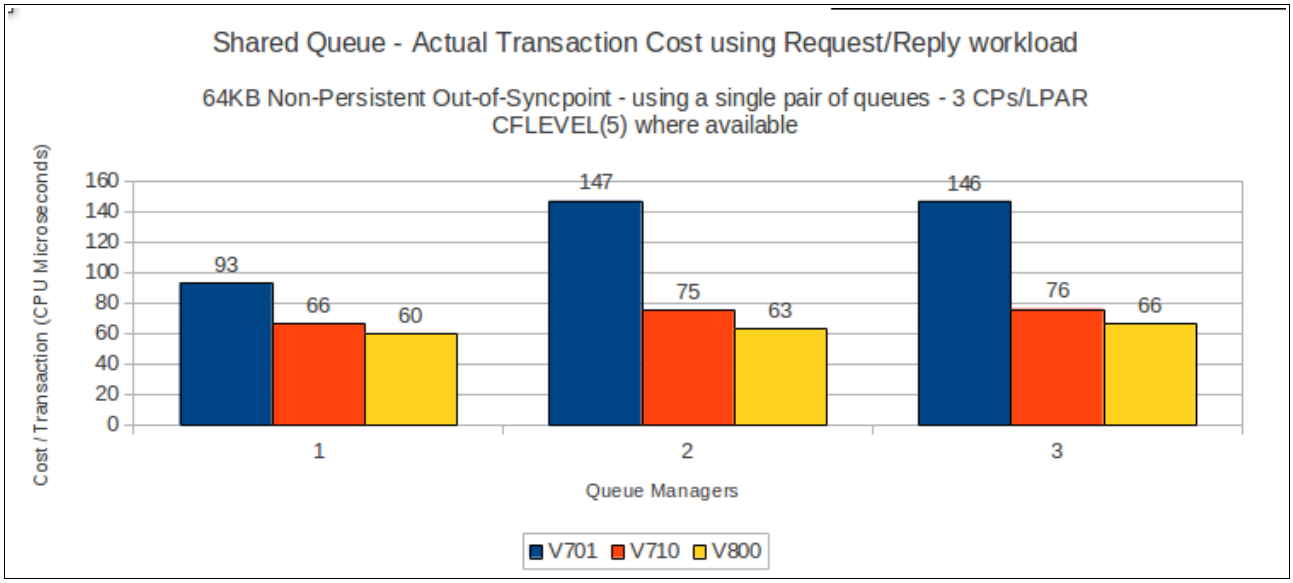
2KB messages



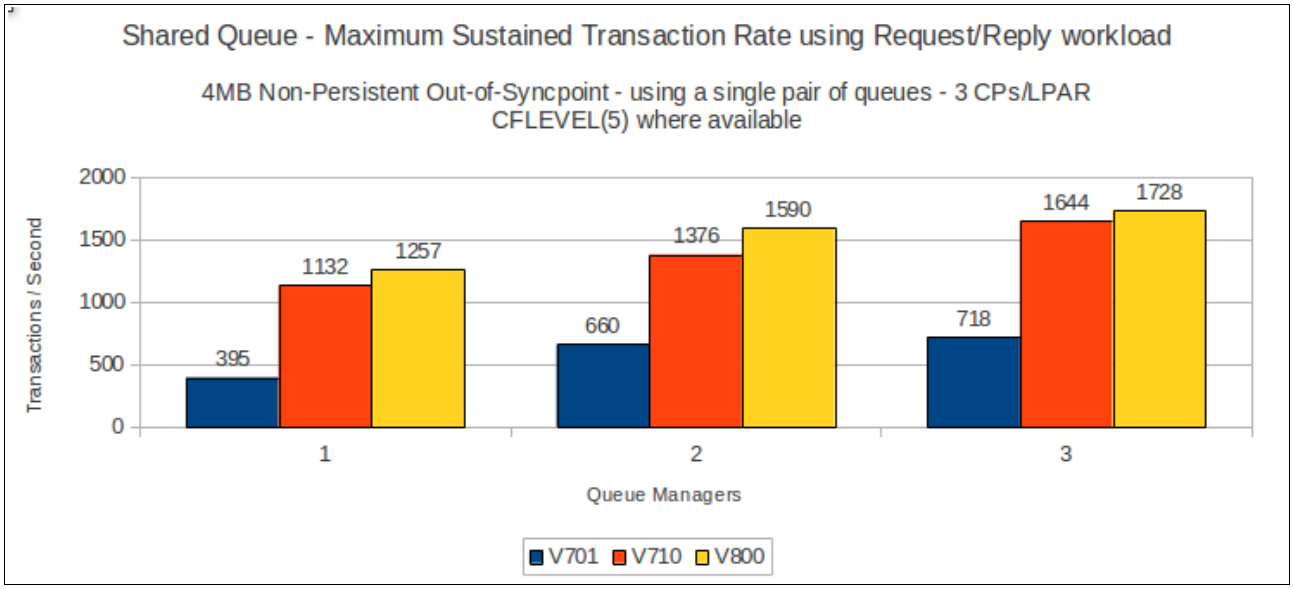


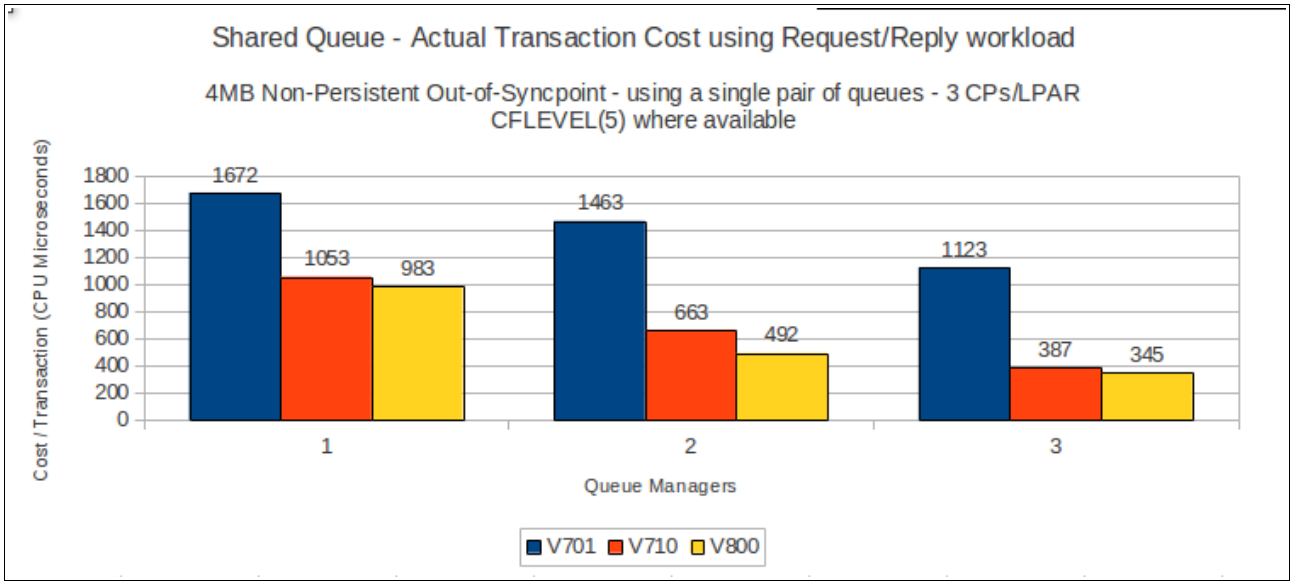
64KB messages





4MB messages





Non-persistent server in-syncpoint workload

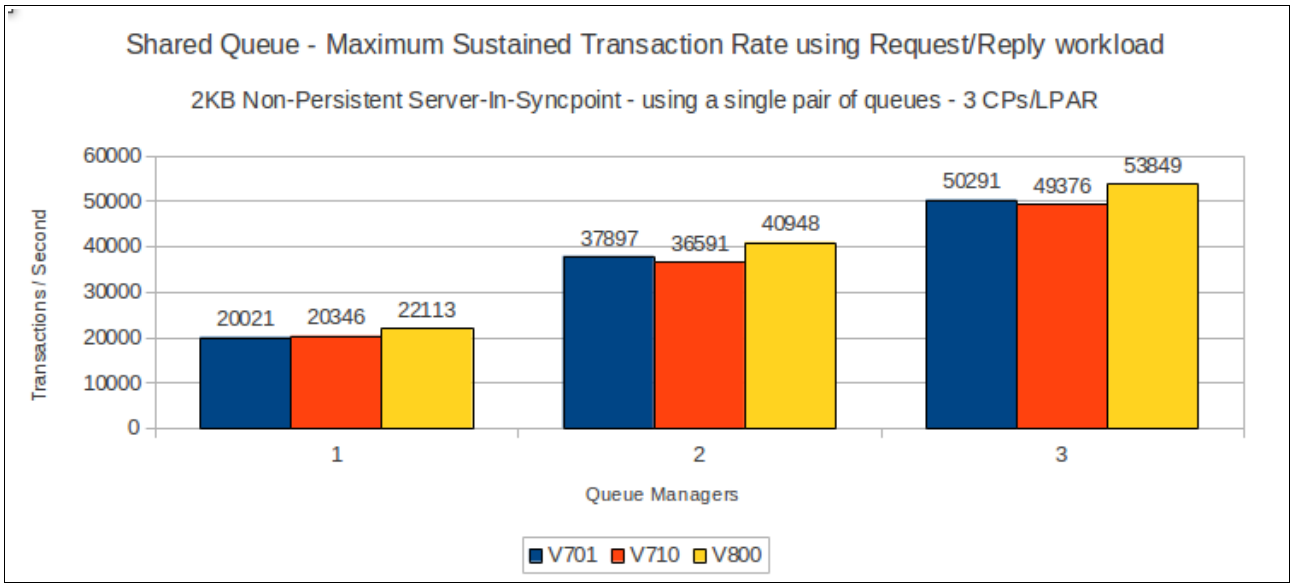
Maximum throughput on single pair of request/reply queues

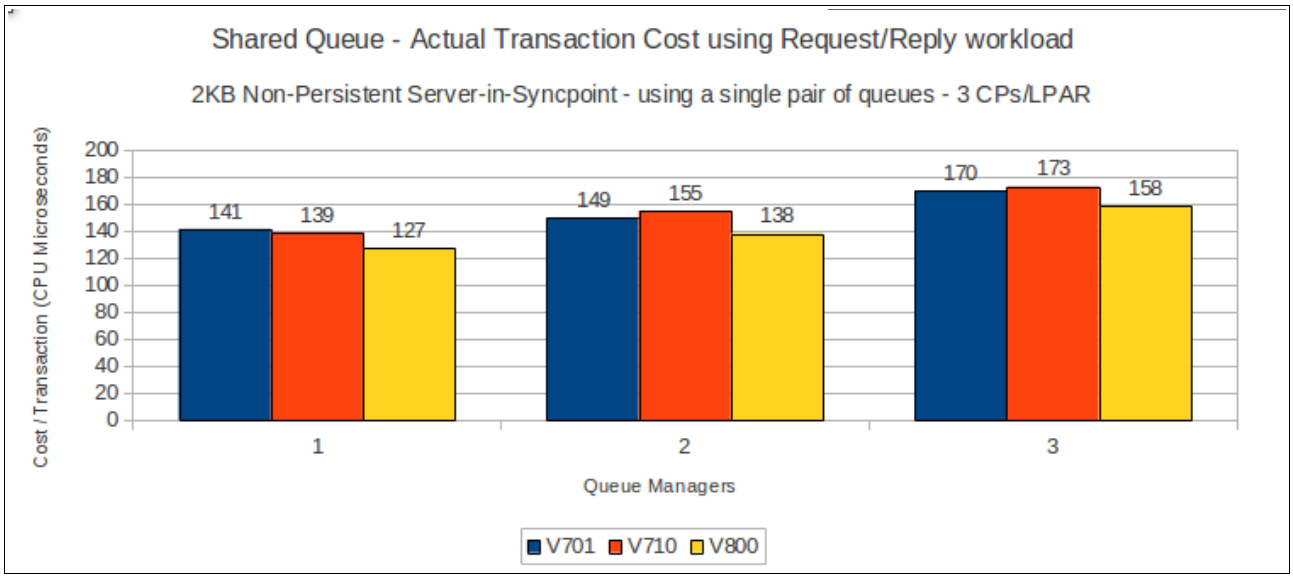
The test uses 5 batch requester tasks that each put a message to a common request queue and wait for a specific response on the reply queue. Once they have got the message, they put another message to the request queue. The messages are put and got out of syncpoint.

There are 4 batch server tasks that action MQGET-with-wait calls on the request queue, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into an MQGET-with-wait. The messages are got and put in-syncpoint.

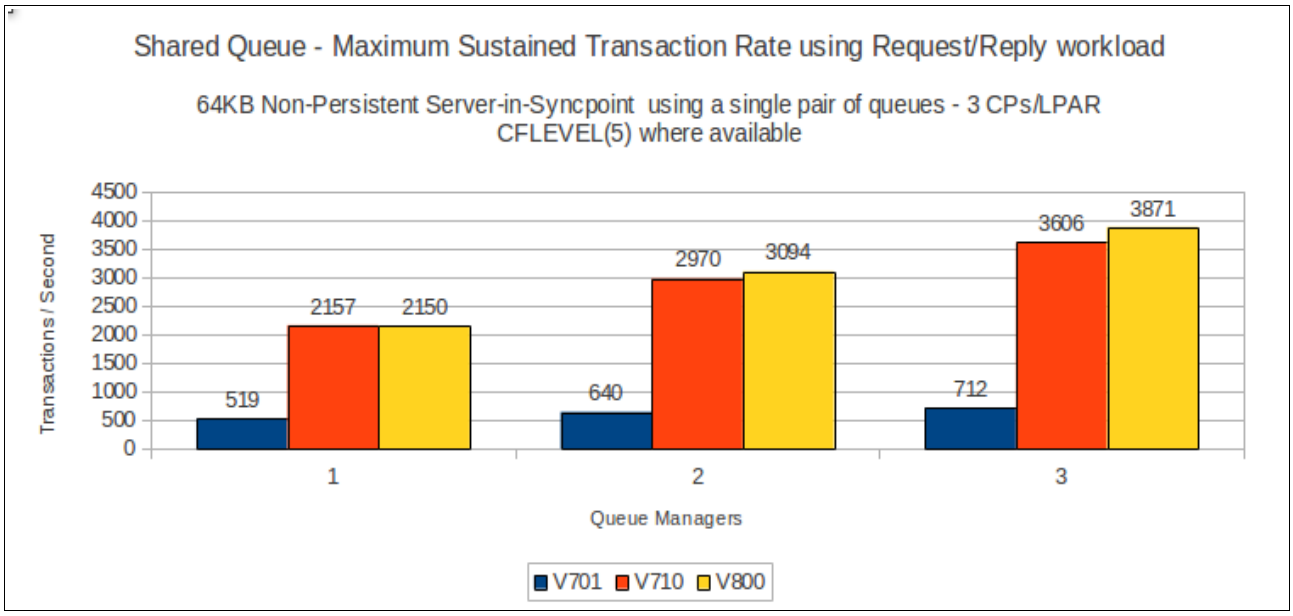
An increasing number of queue managers are allocated to process the workload. Each queue manager has 5 requester and 4 server tasks.

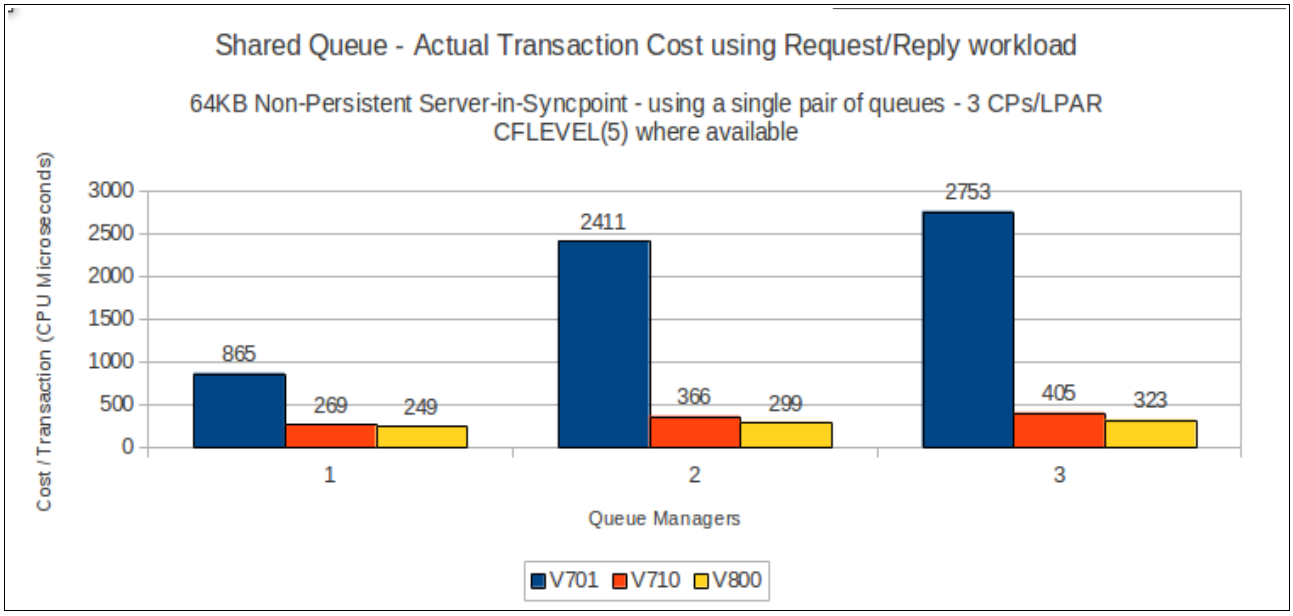
2KB messages



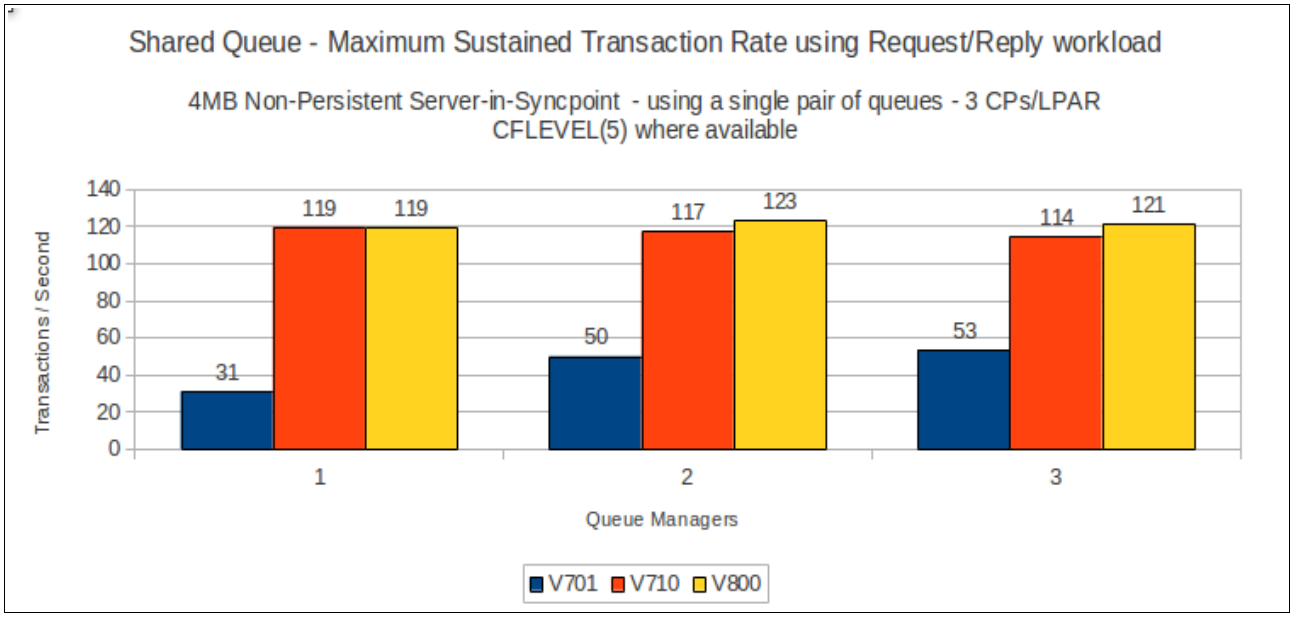


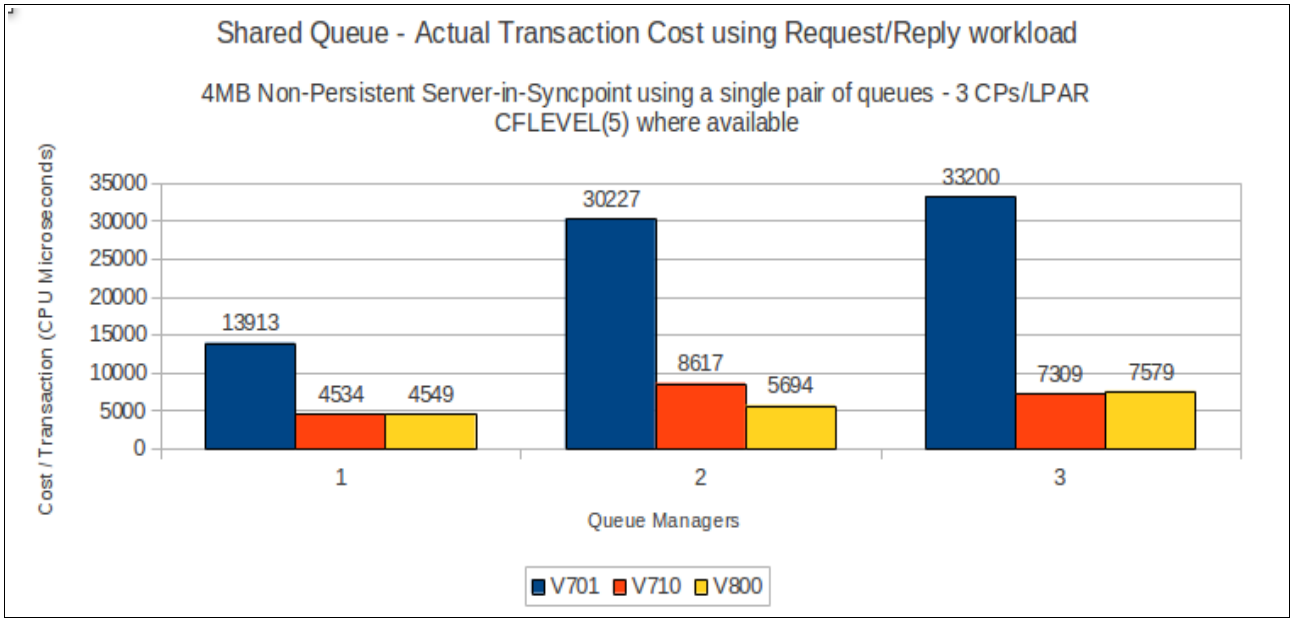
64KB messages





4MB messages – needs updating





Data sharing non-persistent server in-syncpoint workload

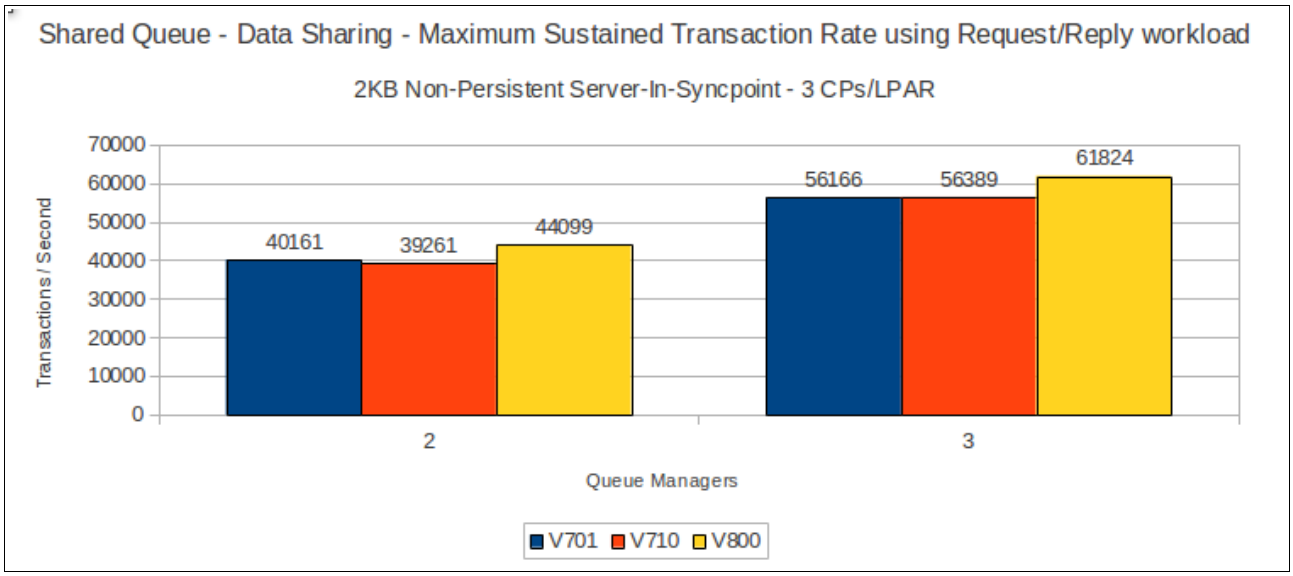
The previous shared queue tests are configured so that any queue manager within the queue sharing group can process messages put by any particular requester application. This means that the message may be processed by a server application on any of the available LPARs. Typically, the message is processed by a server application on the same LPAR as the requester.

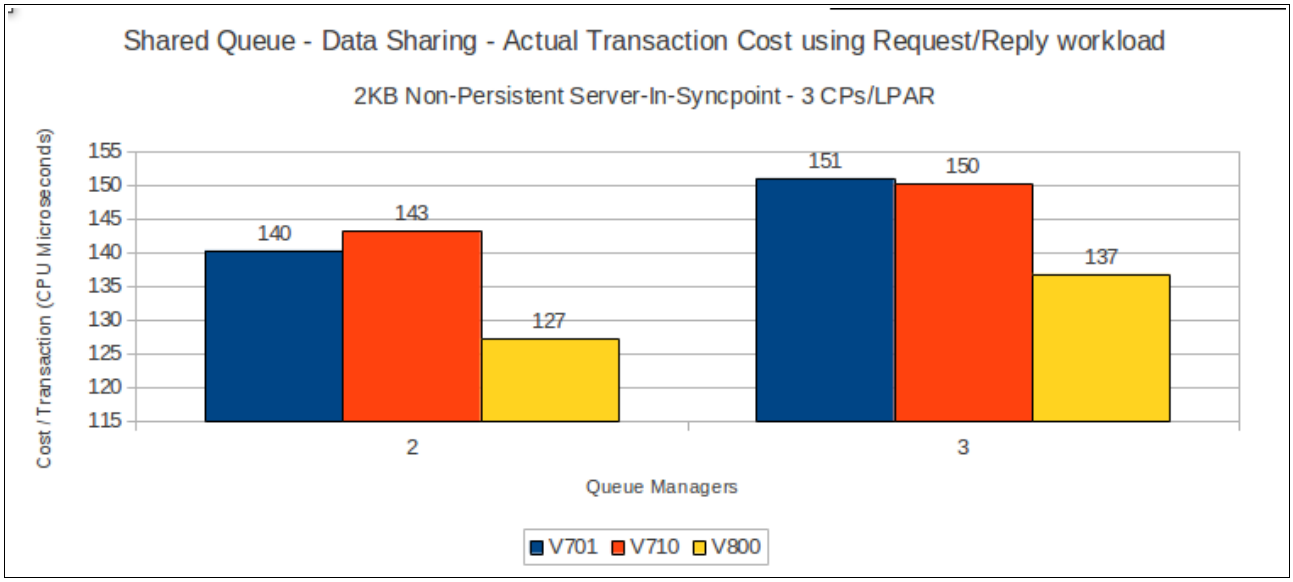
In the following tests, the message can only be processed by a server application on a separate LPAR. This is achieved by using multiple pairs of request/reply queues.

The test uses 5 batch requester tasks on each queue manager that put a message to a specific request queue and wait for a specific response on a specific queue. Once they have got the message, they put another message to the request queue. The messages are put and got out of syncpoint.

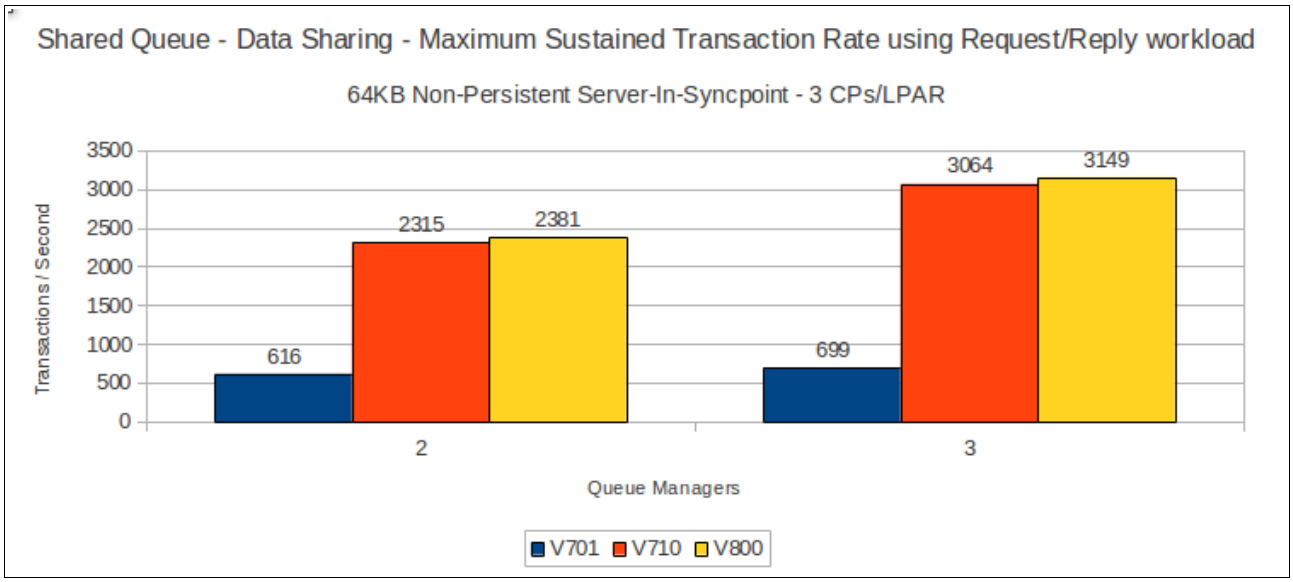
There are 4 batch server tasks on each queue manager that use MQGET-with-wait calls on the request queue of one of the remote queue managers, get the message and put a reply to the known (and pre-opened) reply queue and then the application goes back into an MQGET-with-wait. The messages are got and put in-syncpoint.

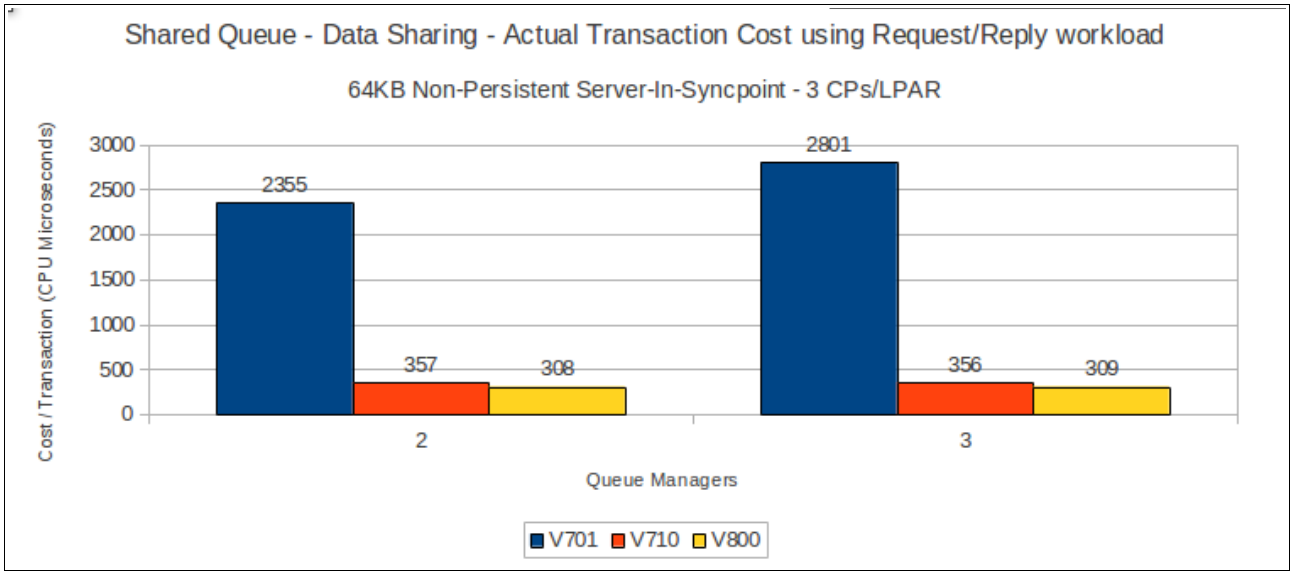
2KB messages





64KB messages





Regression – moving messages across channels

The regression tests for moving messages across channels e.g. sender-receiver channels, are designed to drive the channel initiator such that the network is the constraining factor. Therefore the tests use non-persistent messages out-of-syncpoint, so that they are not constrained by logging capacity, etc.

Within the channel tests there are measurements with small numbers of channels (1 to 4 inbound plus outbound), which was suitable for driving the network to capacity and also tests with 10 to 50 channels.

For each of the test types, the channels are used both with and without SSL encryption enabled.

Unless otherwise stated, the SSL tests are configured to use “TRIPLE_DES_SHA_US” encryption on the Crypto-accelerator processor. The SSLRKEYC attribute is set so that 1MB of data can flow across the channel before renegotiating the key.

New in this performance report is the inclusion of a comparison of channel compression performance. As mentioned earlier in the document, version 8.0 can exploit zEDC hardware compression, but to allow for a more simple release on release comparison, all these performance measurements will only use software compression.

In addition, compression on SSL channels has also been compared using one of the newer encryption ciphers “ECDHE_RSA_AES_256_CBC_SHA384” , which is only available on version 7.1 (via APAR PM77341) and version 8.0.

For further guidance on channel tuning and usage, please refer to the supportpac MP16 – Capacity Planning and Tuning Guide.

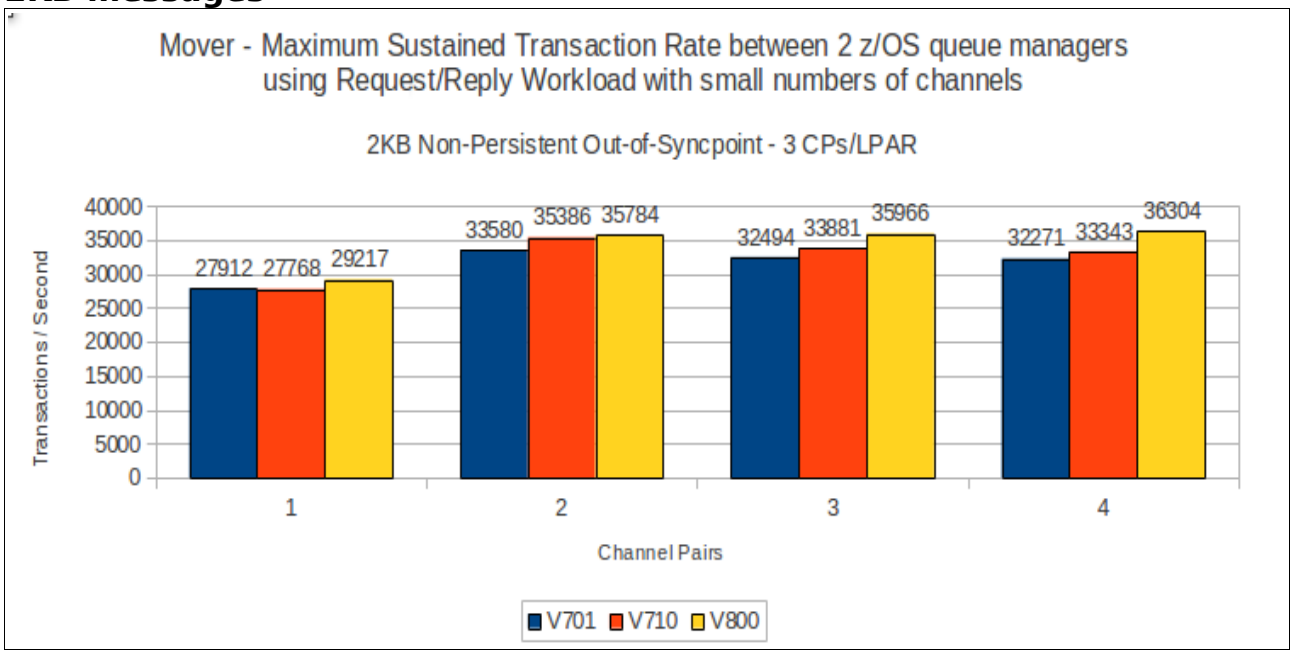
The measurements using 1 to 4 channels use batch applications to drive the workload at each end of the channel.

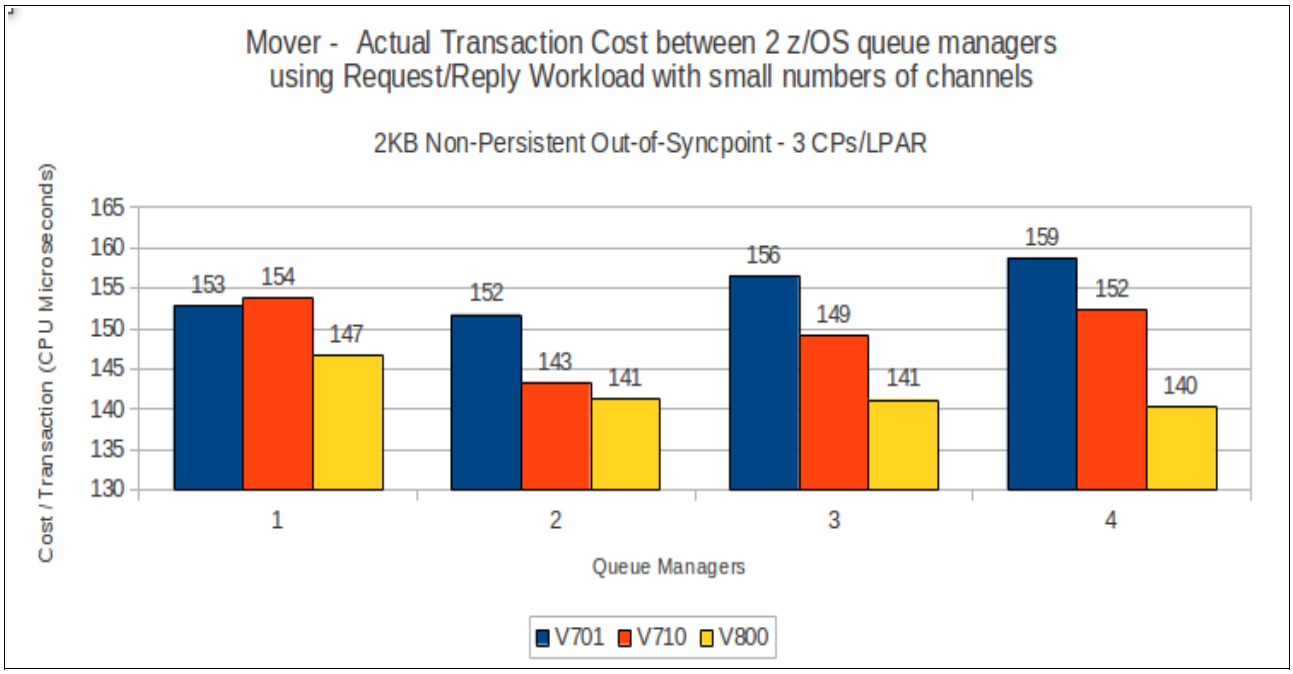
The measurements using 10 to 50 channels use long-lived CICS transactions to drive the workload. This means that each CICS application will put and get thousands of messages before ending. This model means that we are not including the cost of starting a CICS transaction, opening and closing queues and the teardown of the transaction at the end of the workload.

The compression tests use 64KB messages of varying compressibility, so there is something to compress, e.g. a message of 64KB that is 80% compressible would reduce to approximately 13KB. The tests are run using 1 outbound and 1 inbound channel so include the compression and inflation costs for the request and the reply message.

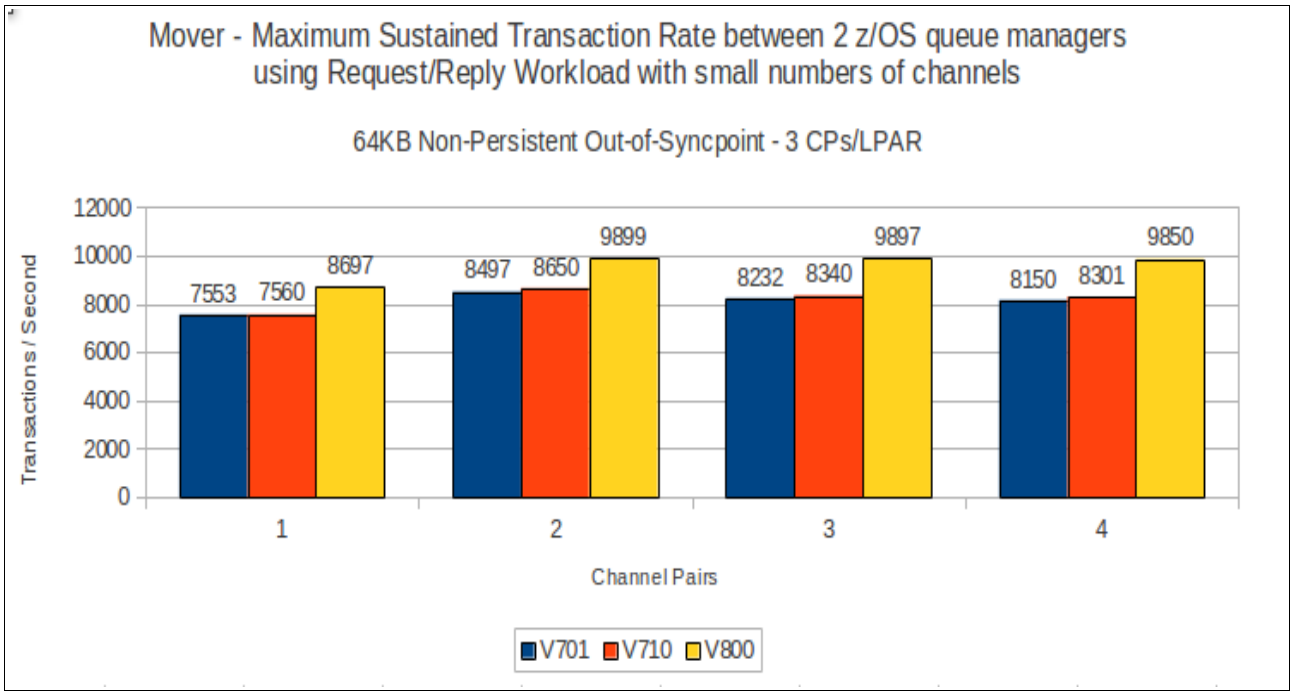
Non-persistent out-of-syncpoint – 1 to 4 sender-receiver channels

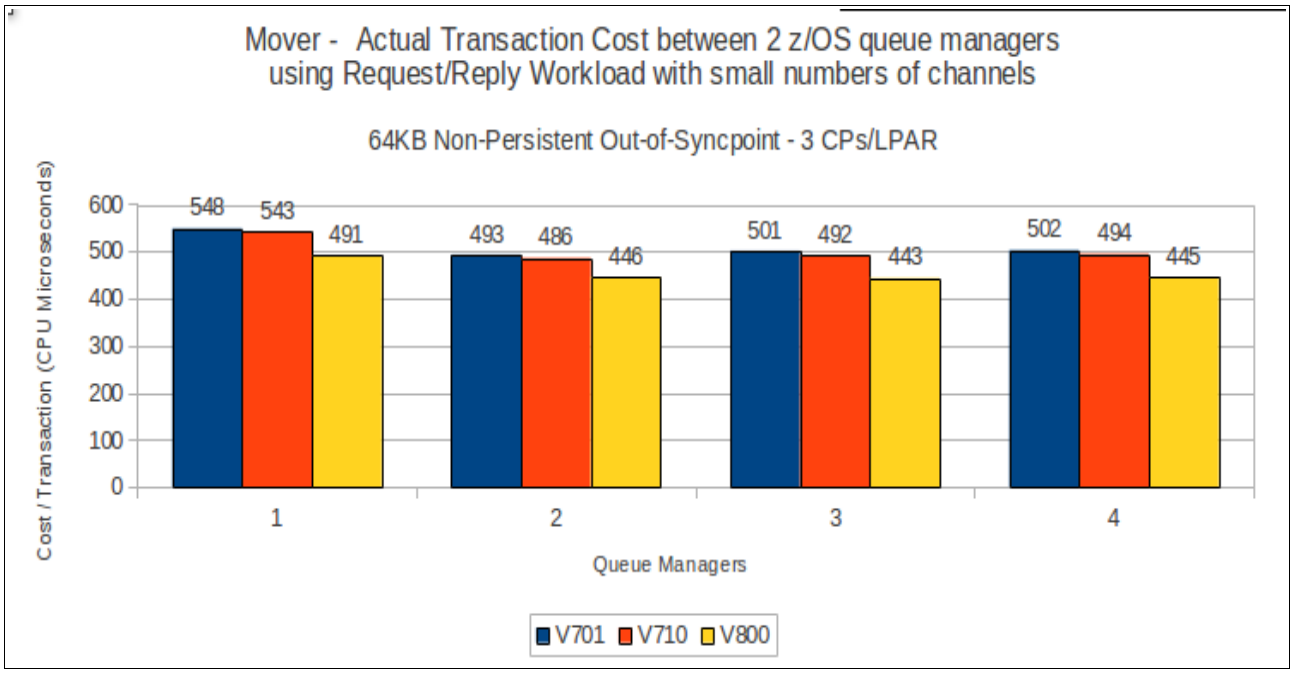
2KB messages



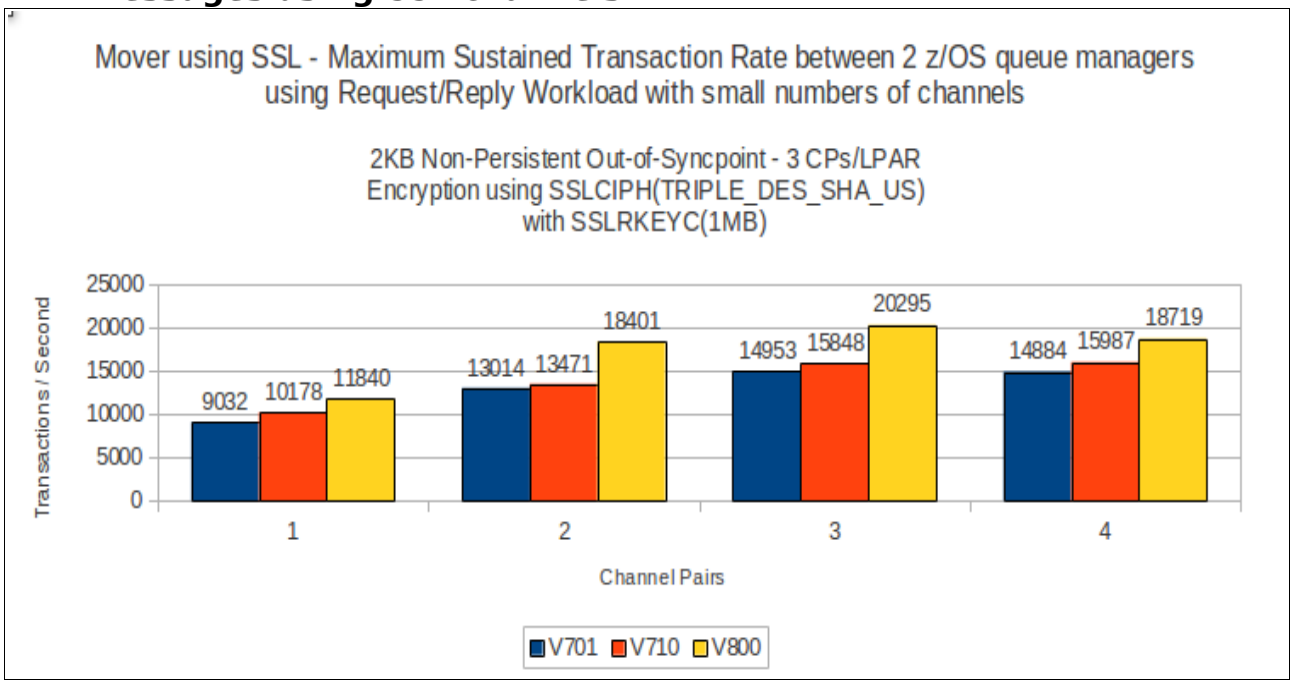


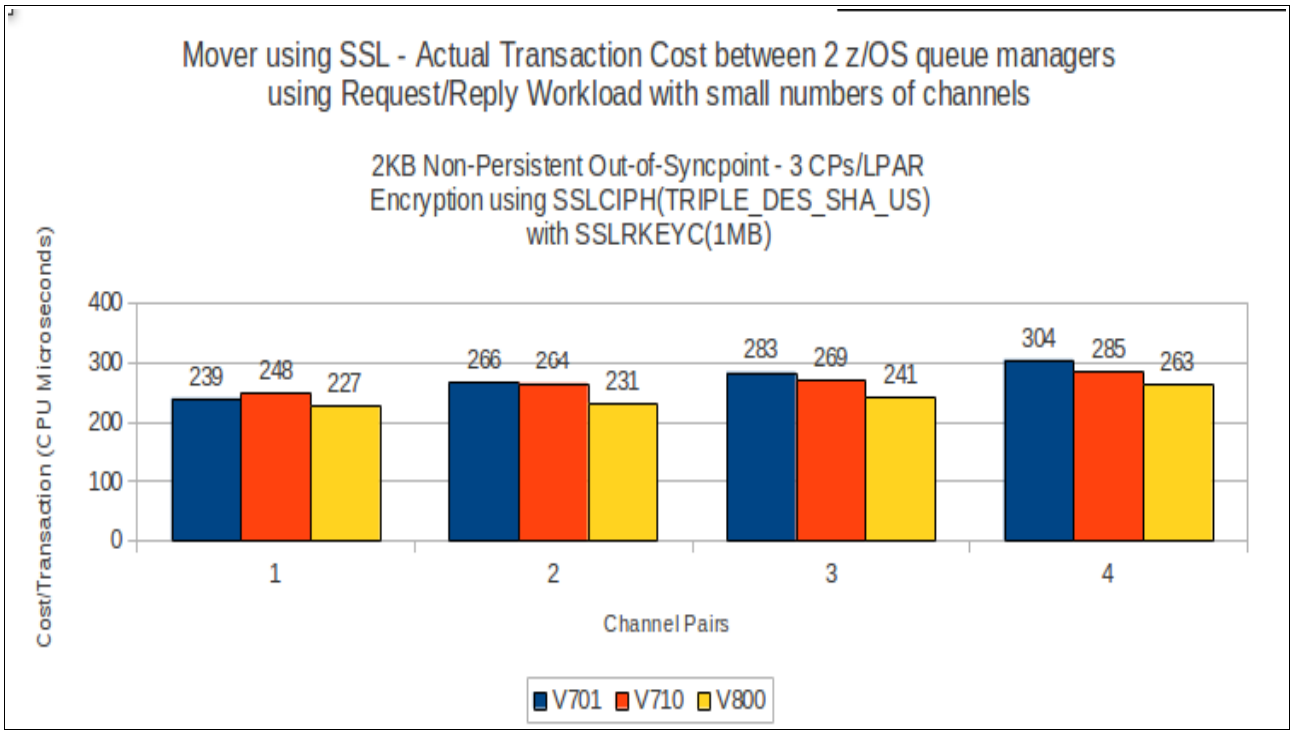
64KB messages



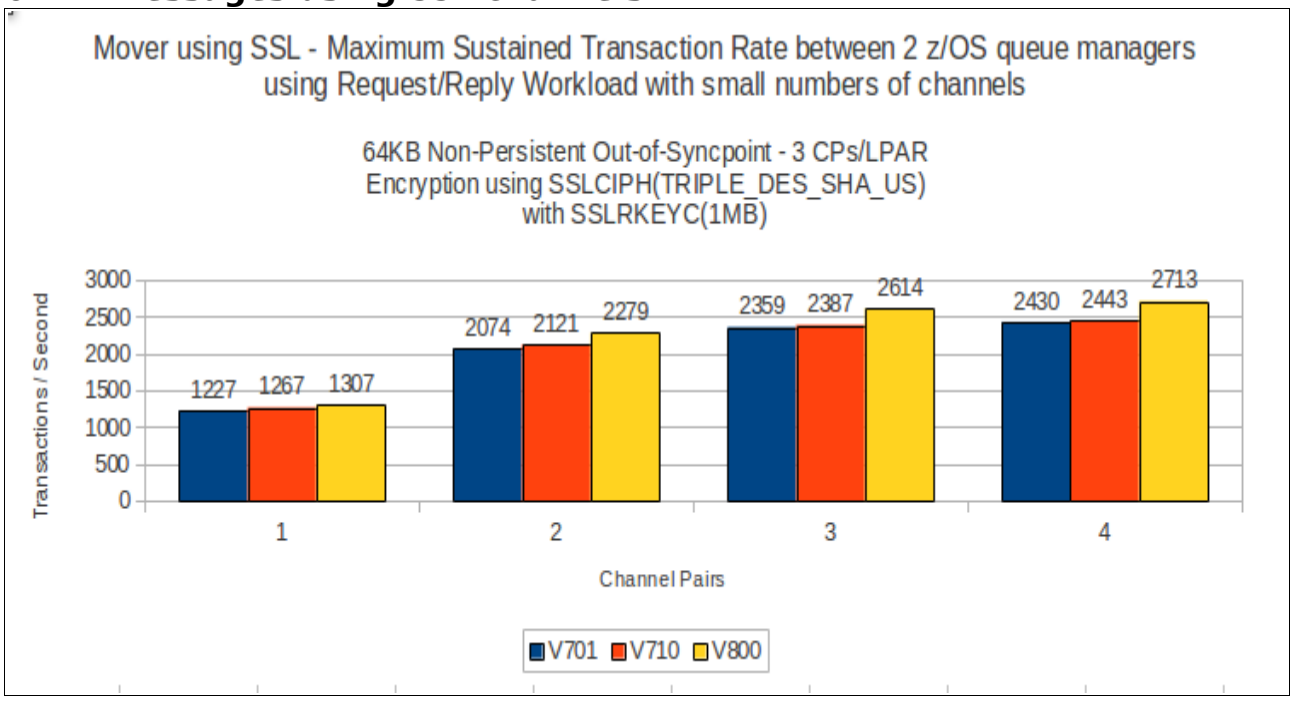


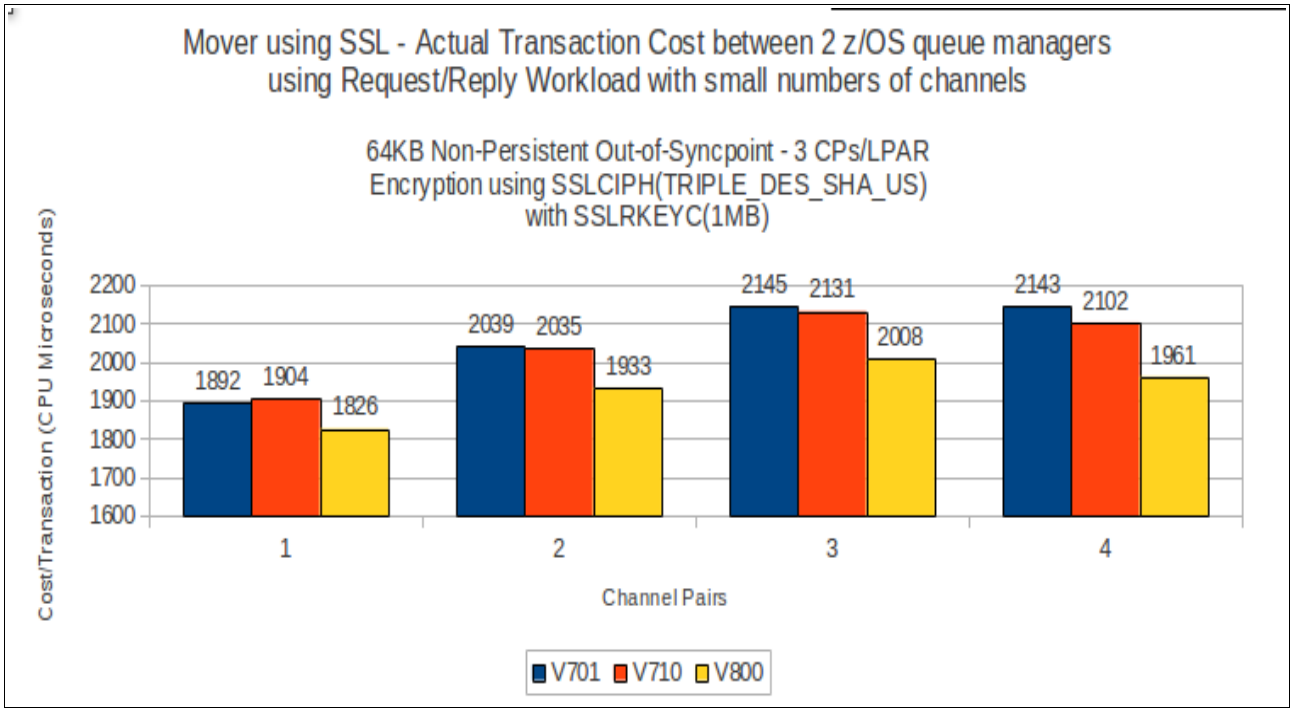
2KB messages using SSL channels





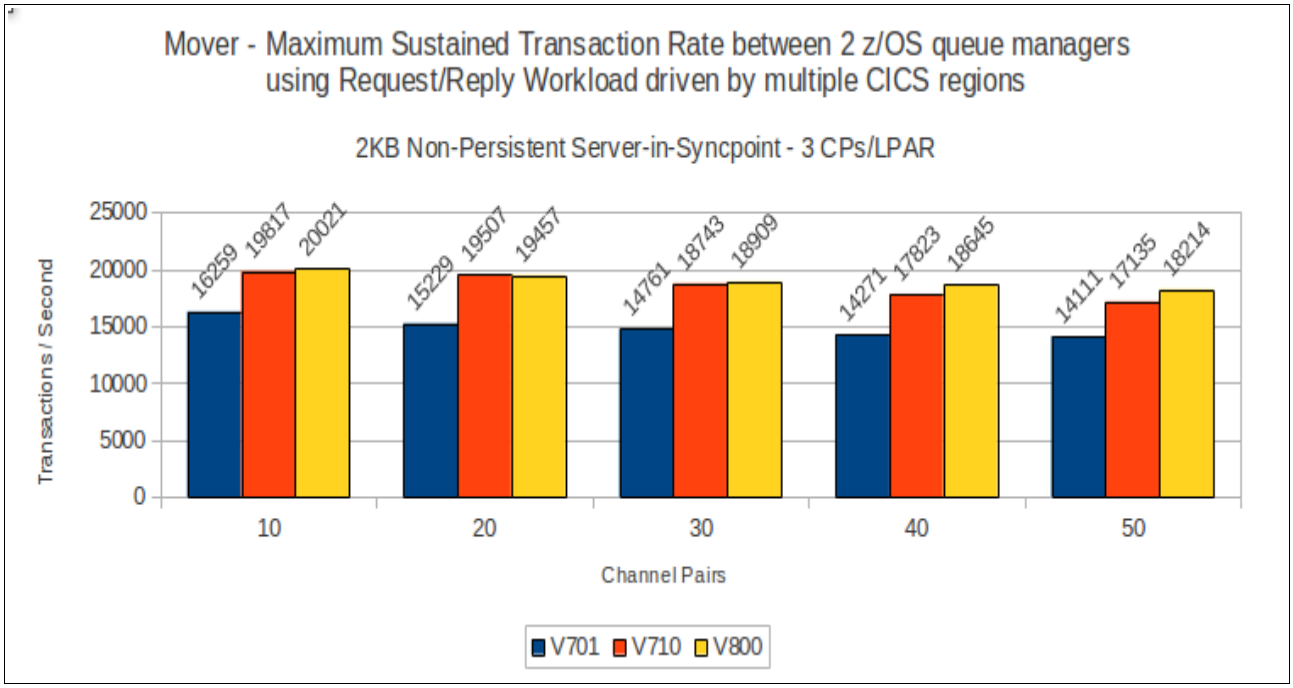
64KB messages using SSL channels

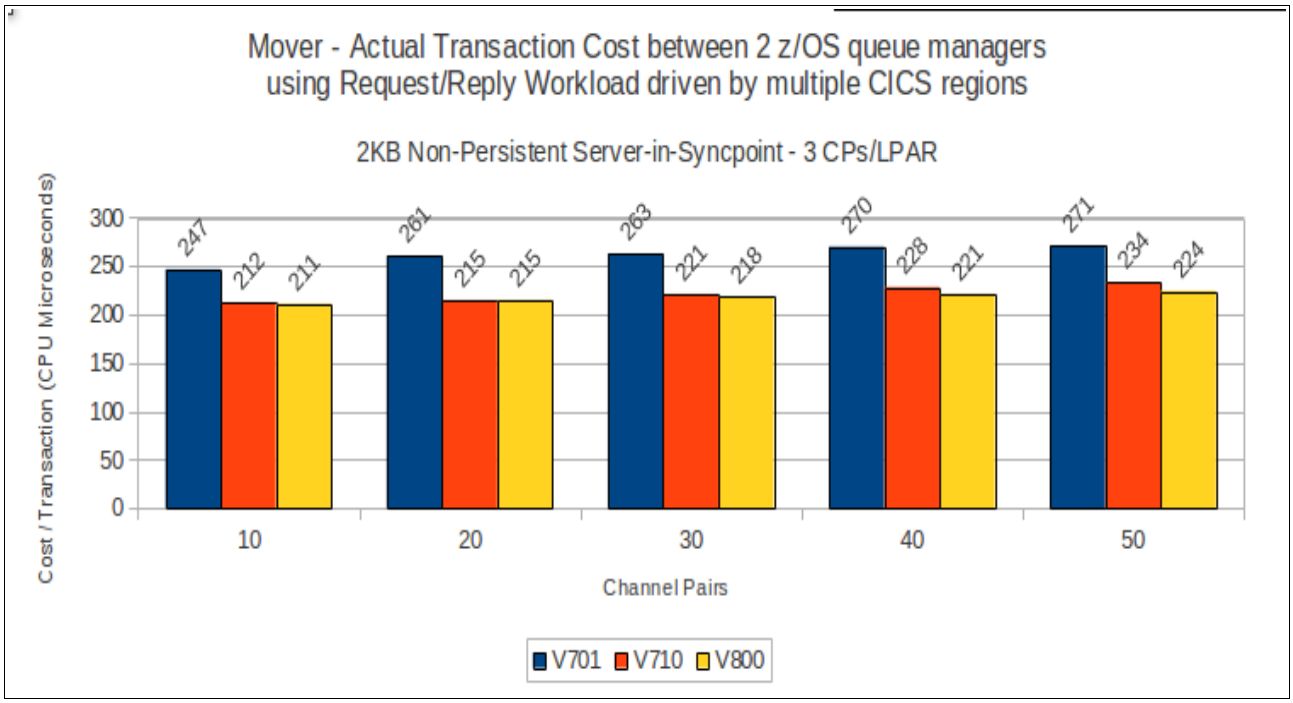




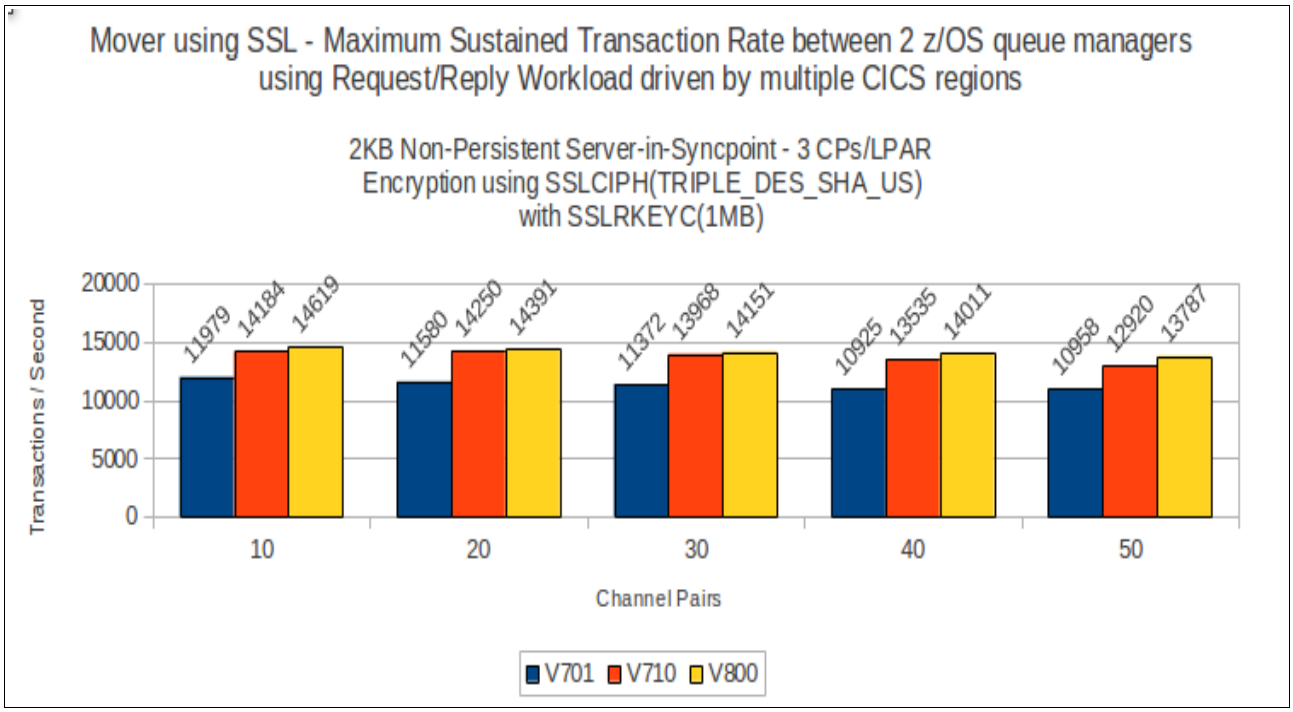
Non-persistent out-of-syncpoint – 10 to 50 sender-receiver channels

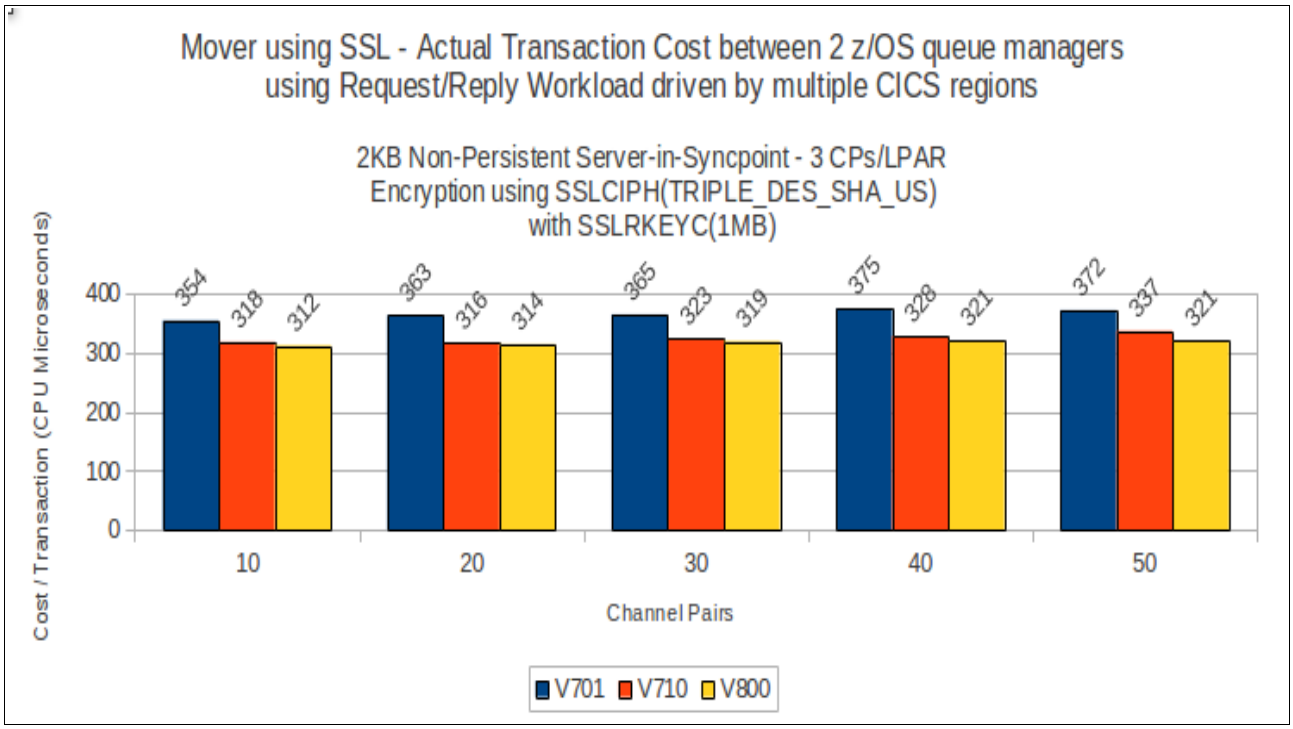
2KB messages





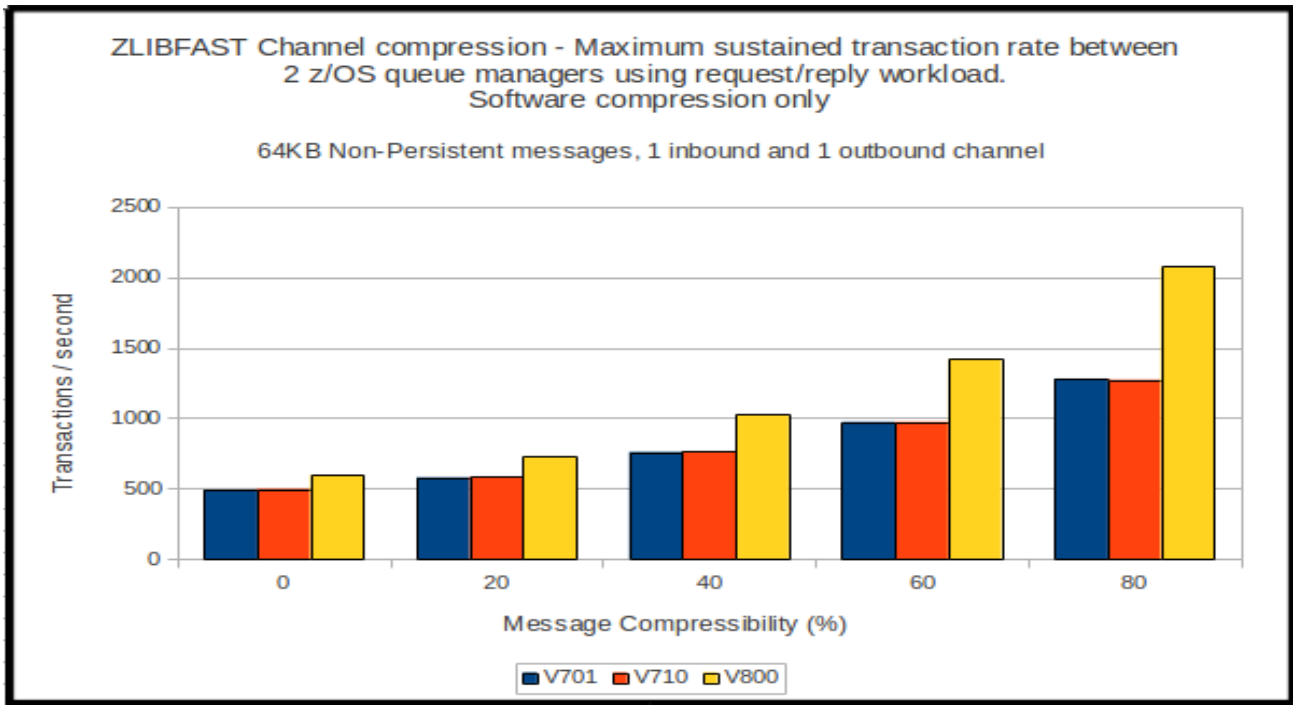
2KB messages using SSL channels

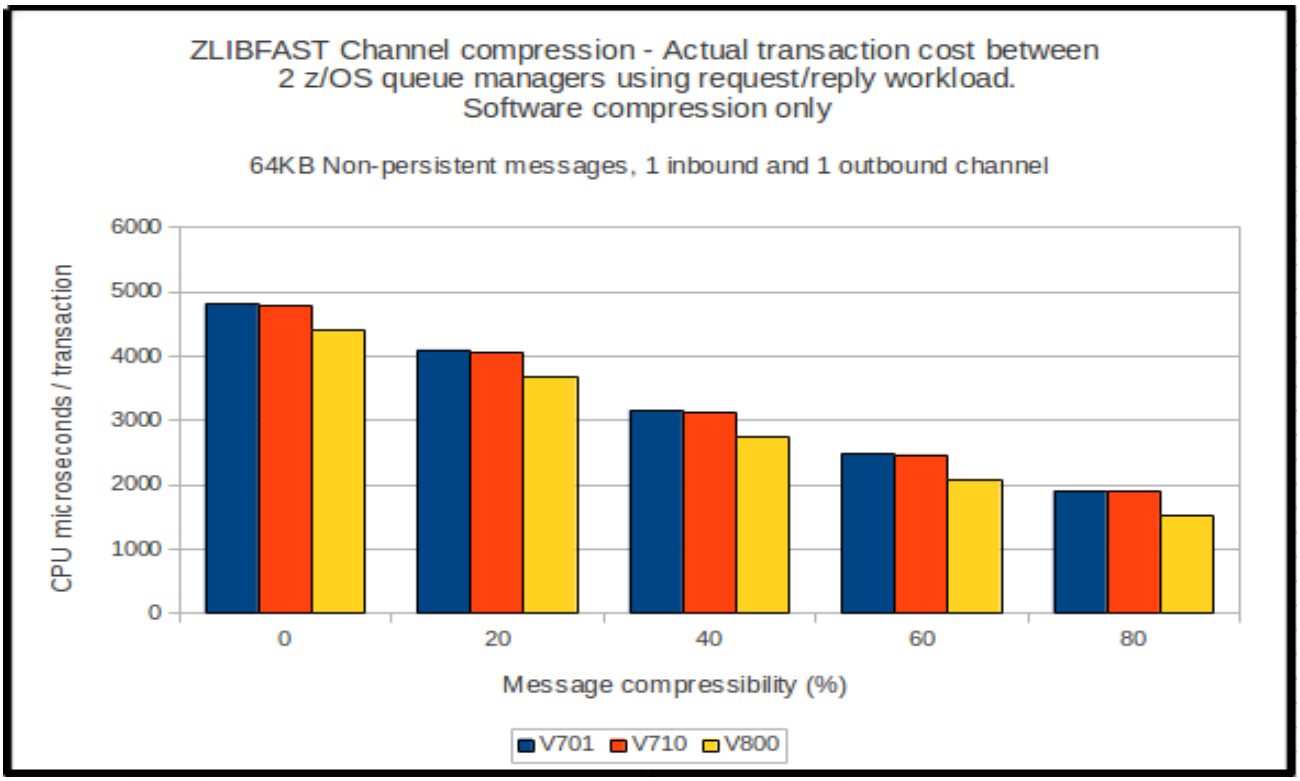




64KB Channel Compression using ZLIBFAST (software only)

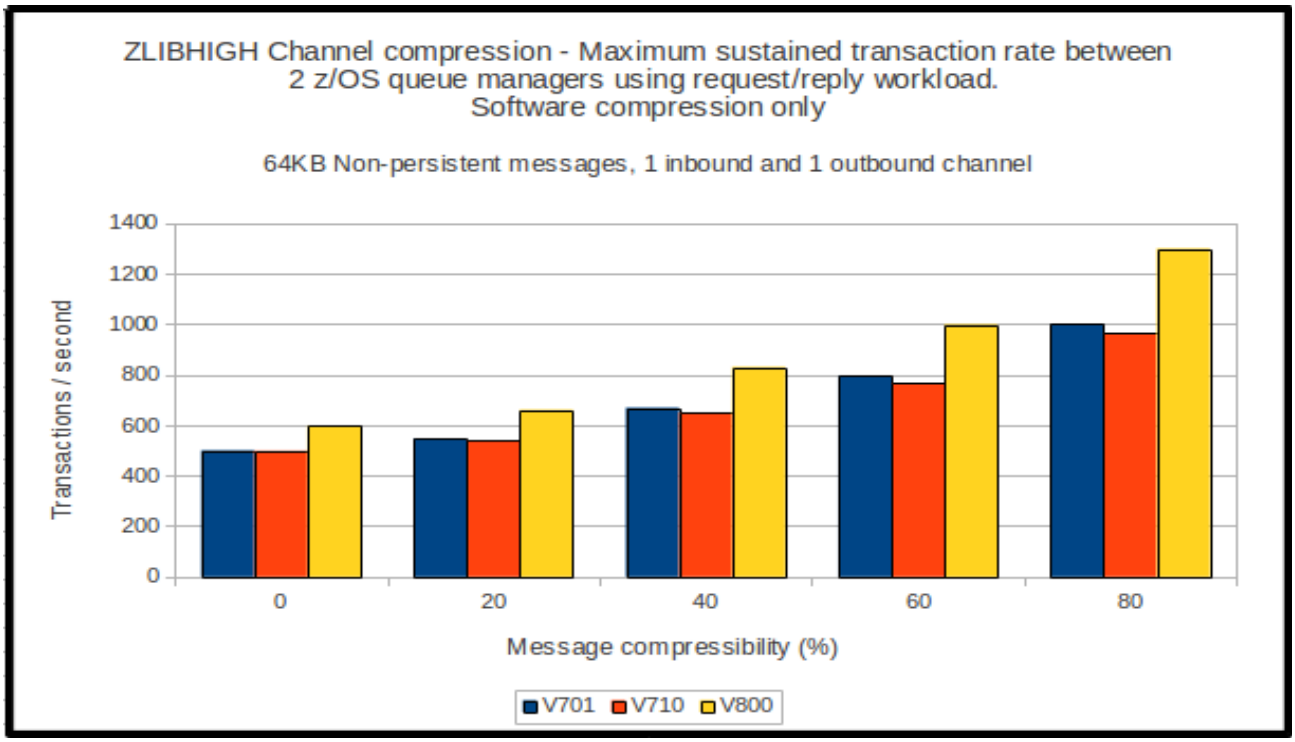
For ZLIBFAST we have seen between a 10 and 20% reduction in cost in the channel initiator between versions 7.0 and 7.1 with version 8.0.

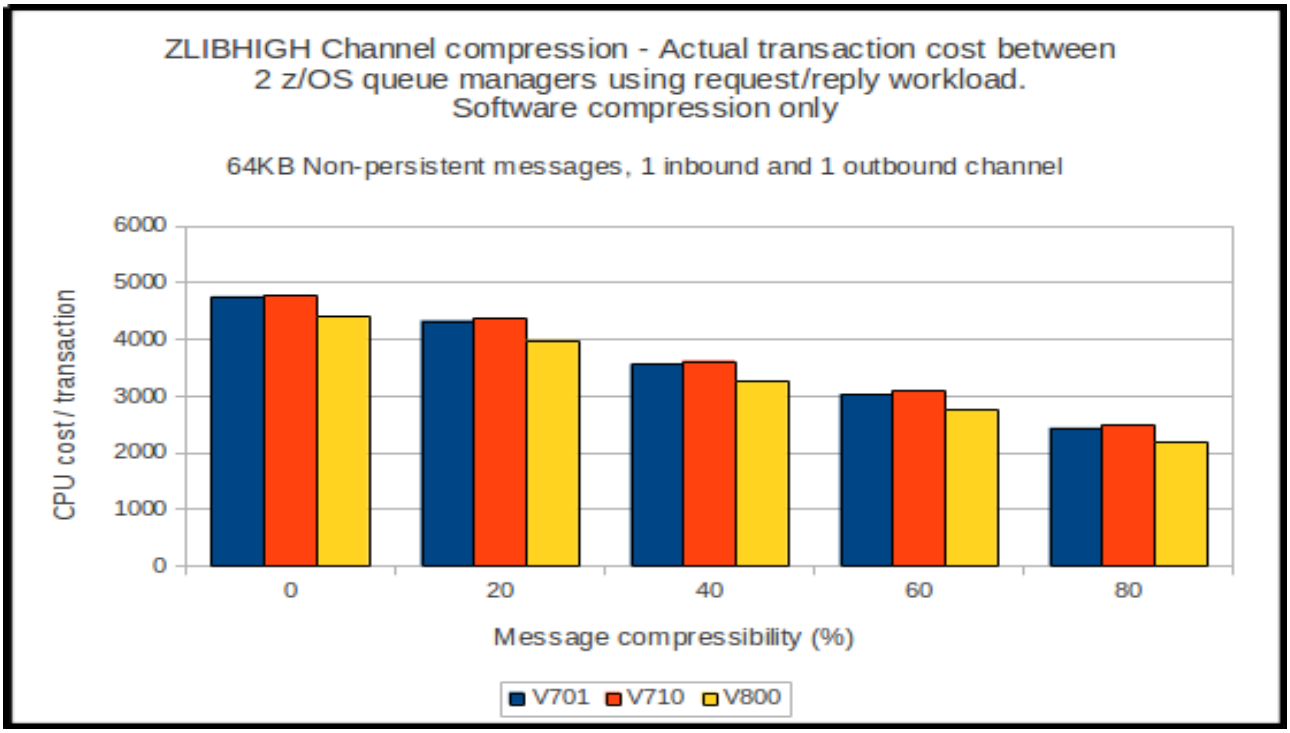




64KB Channel Compression using ZLIBHIGH

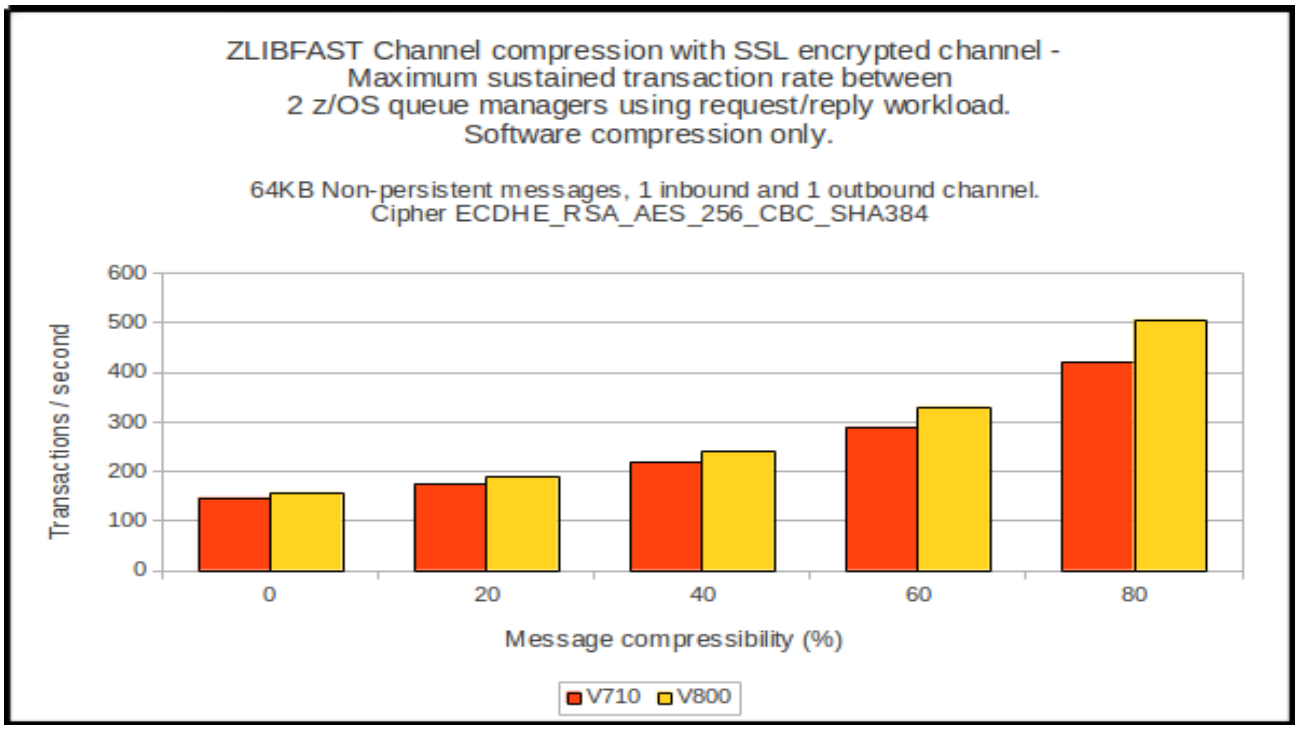
For ZLIBHIGH we have seen between a 5 and 15% reduction in cost in the channel initiator between versions 7.0 and 7.1 with version 8.0.

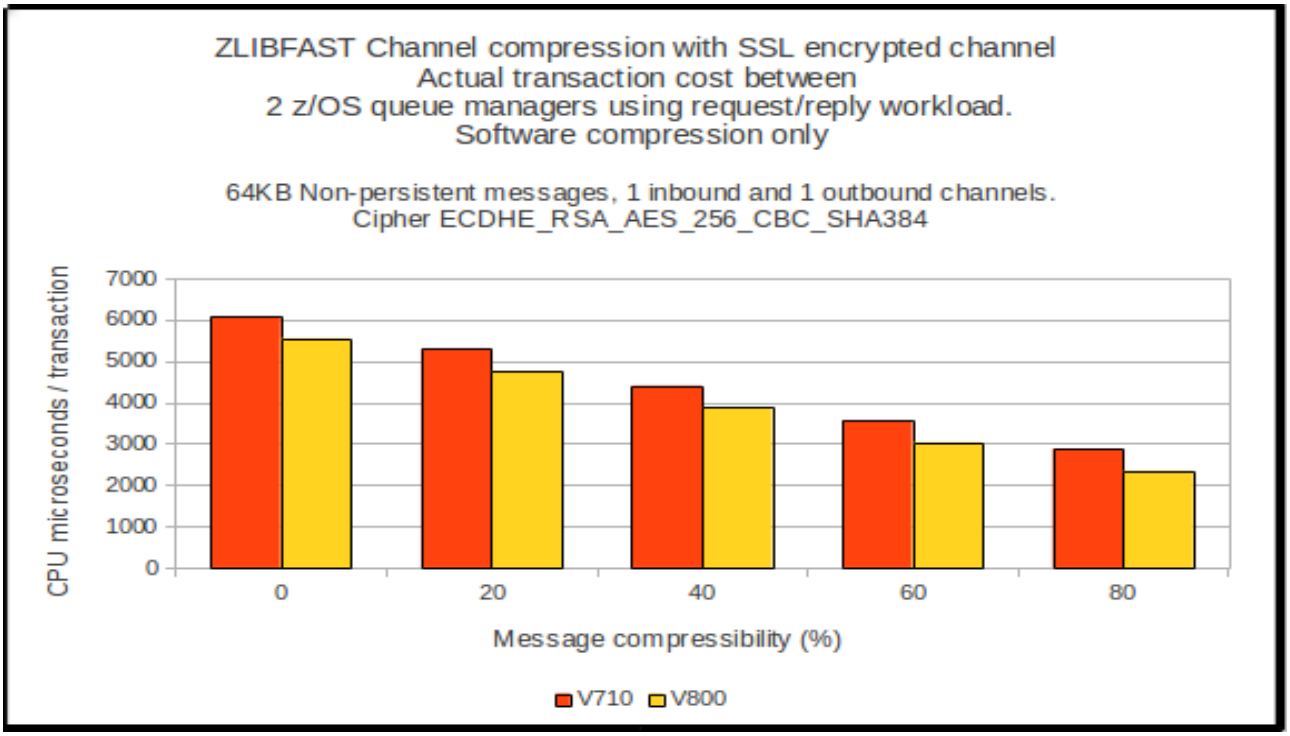




64KB Channel Compression using ZLIBFAST with SSL channels

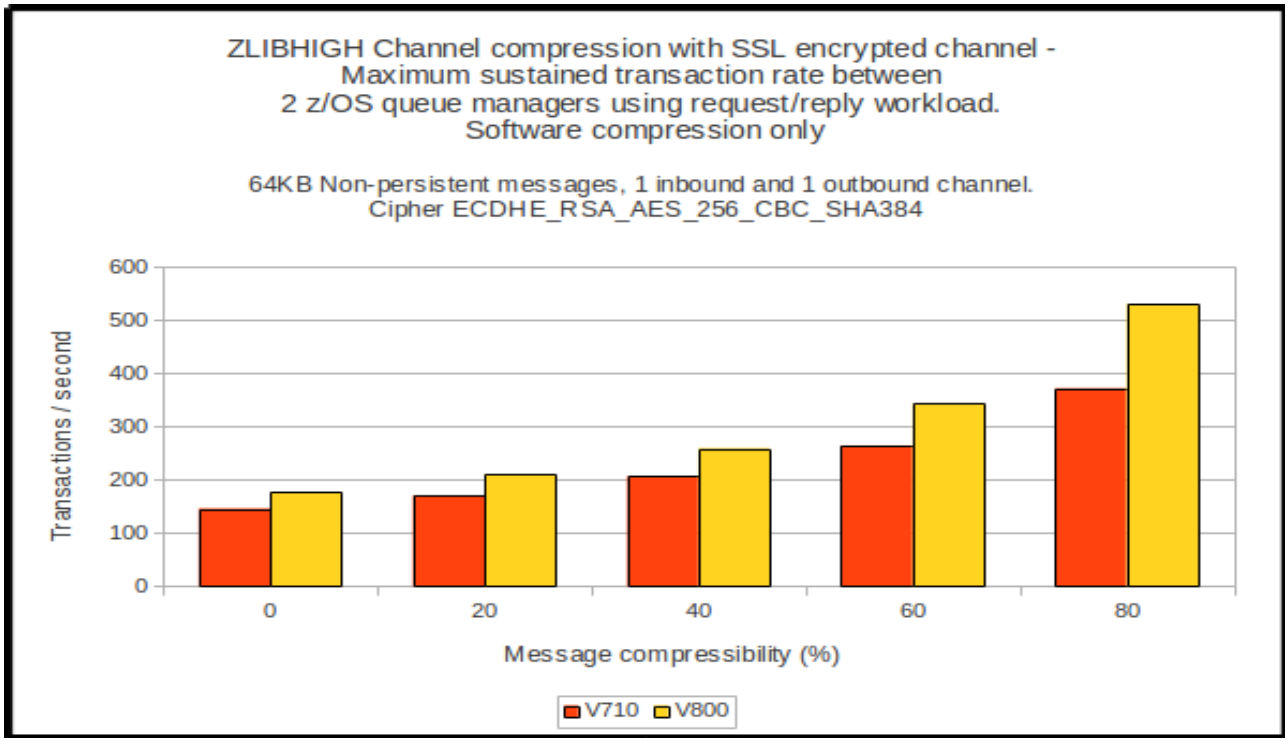
For ZLIBFAST we have seen between a 10 and 20% reduction in cost in the channel initiator between version 7.1 with version 8.0.

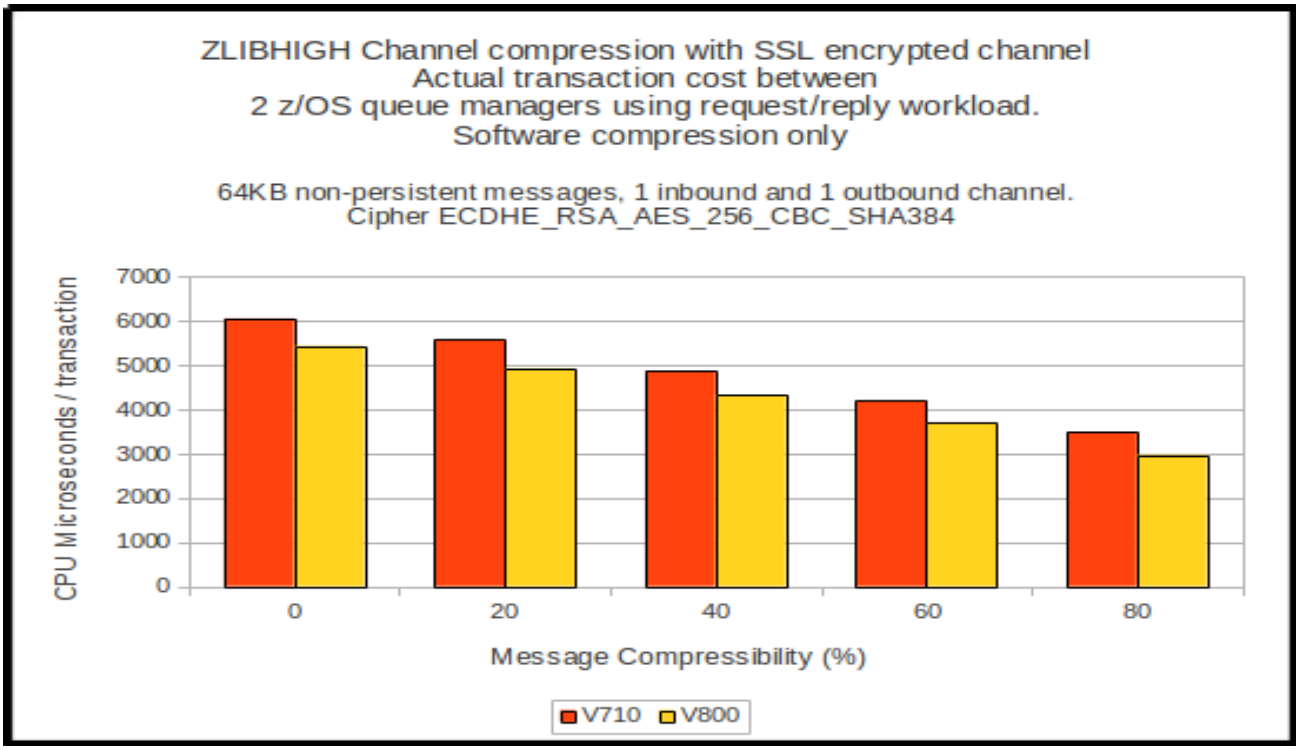




64KB Channel Compression using ZLIBHIGH with SSL channels

For ZLIBHIGH we have seen between a 5 and 15% reduction in cost in the channel initiator between versions 7.1 with version 8.0.





Regression – moving messages across cluster channels

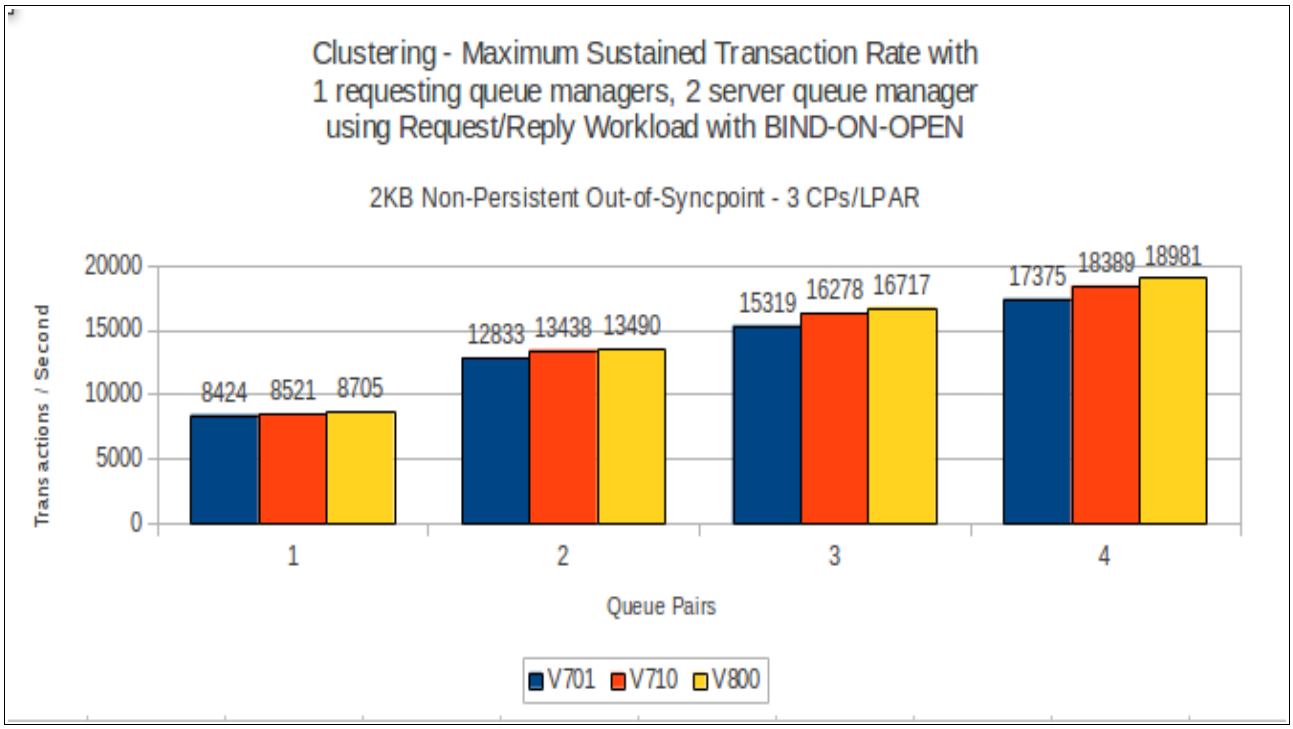
The regression tests for moving messages across cluster channels are relatively simple, providing multiple destinations for each message put.

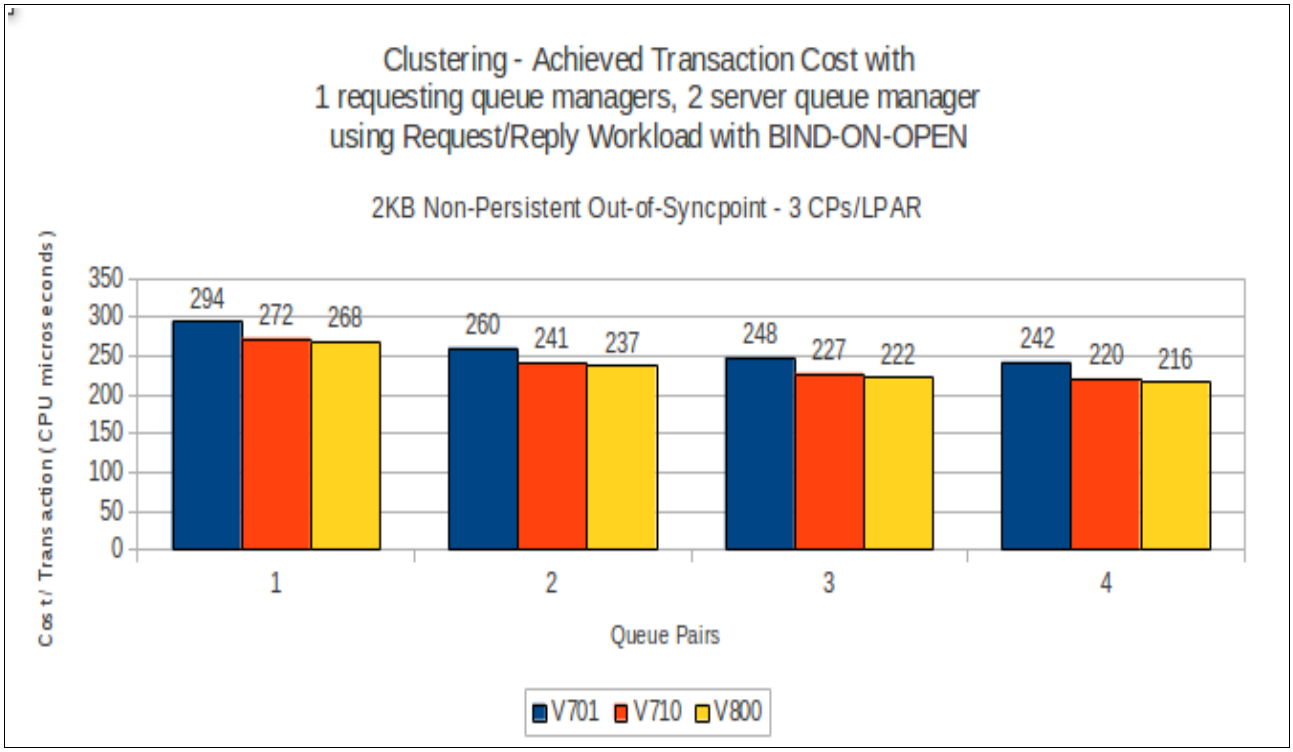
The cluster has 3 queue managers – one for the requester workload and two for the server workload. The queue managers hosting the server workload are both full repository queue managers.

A set of requester tasks is run against one queue manager and the application chooses either bind-on-open or bind-not-fixed. The messages flow across the cluster channels to one of the server queue managers to be processed by the server applications, at which point they are returned to the originating requesting applications.

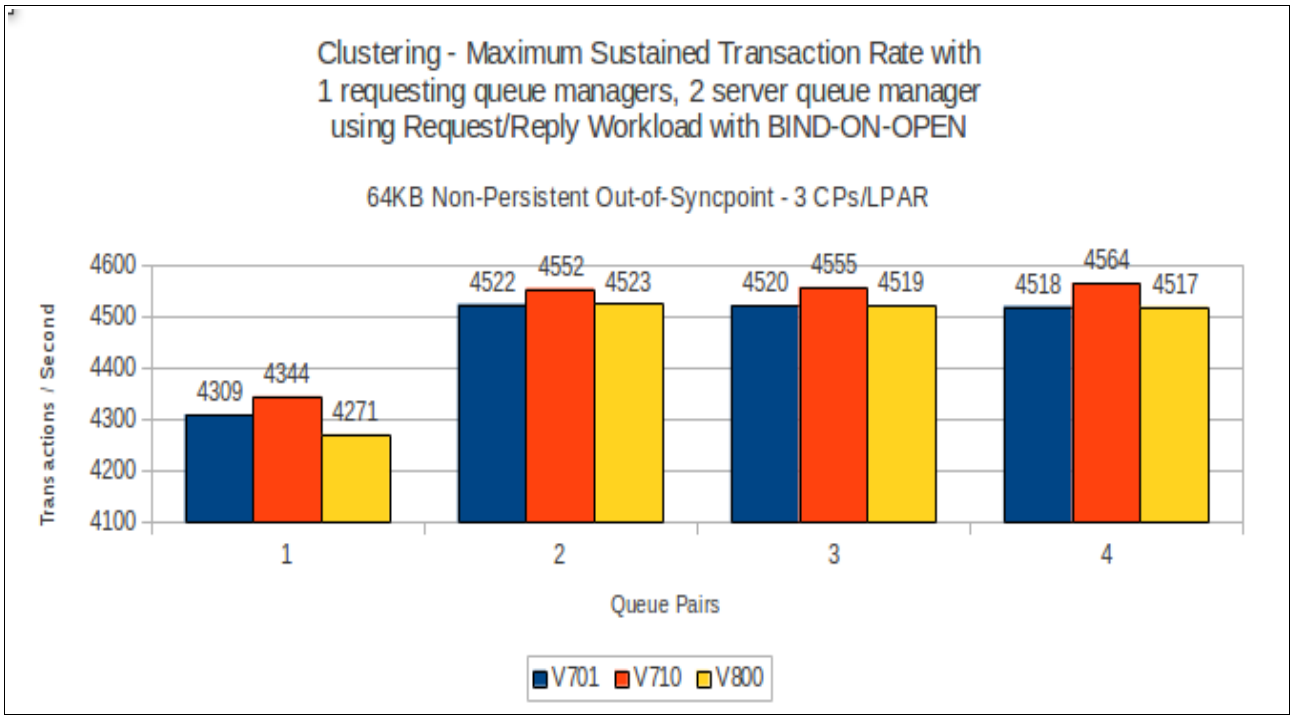
An increasing number of queues (with an corresponding increase in applications) are used.

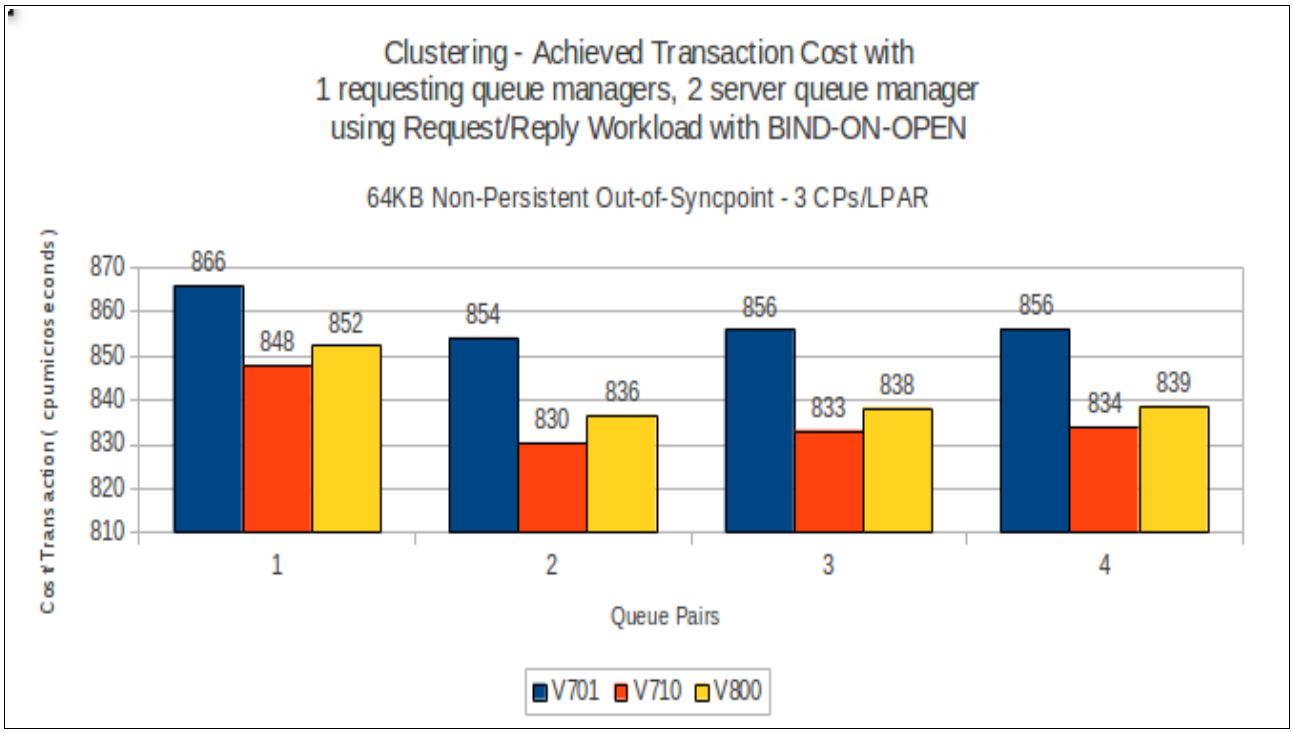
2KB messages – bind-on-open



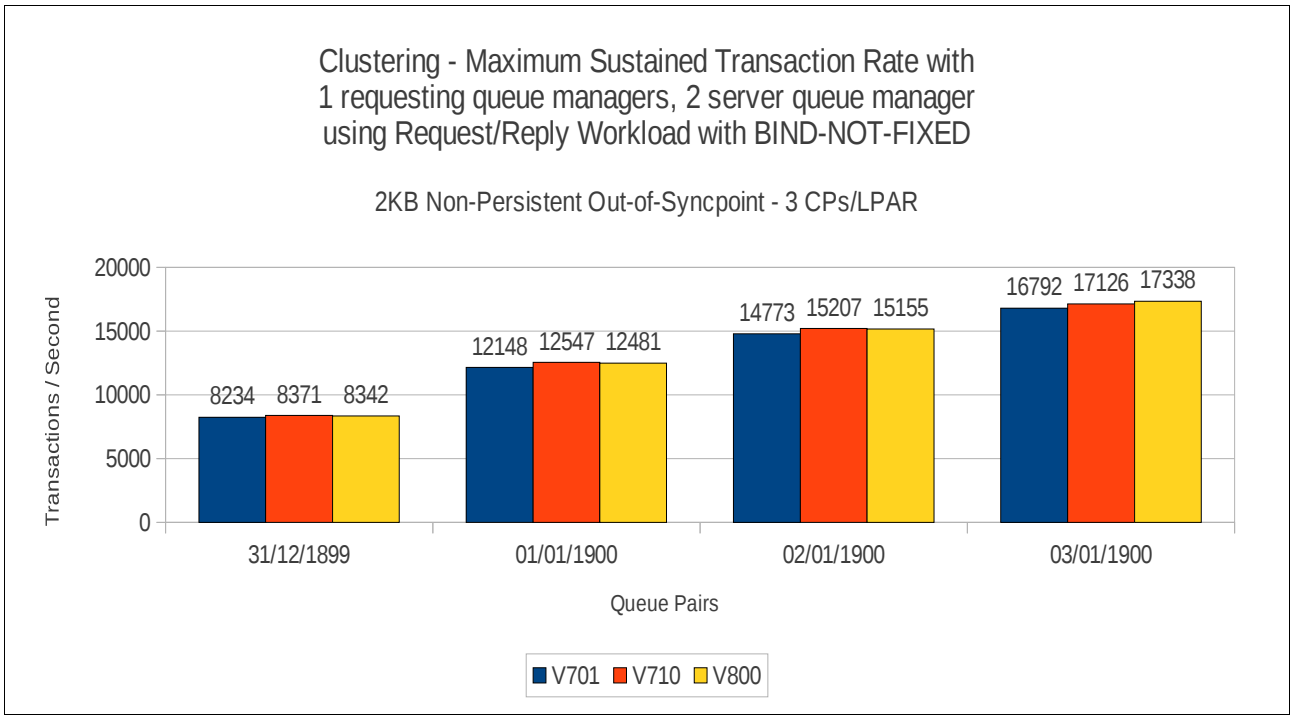


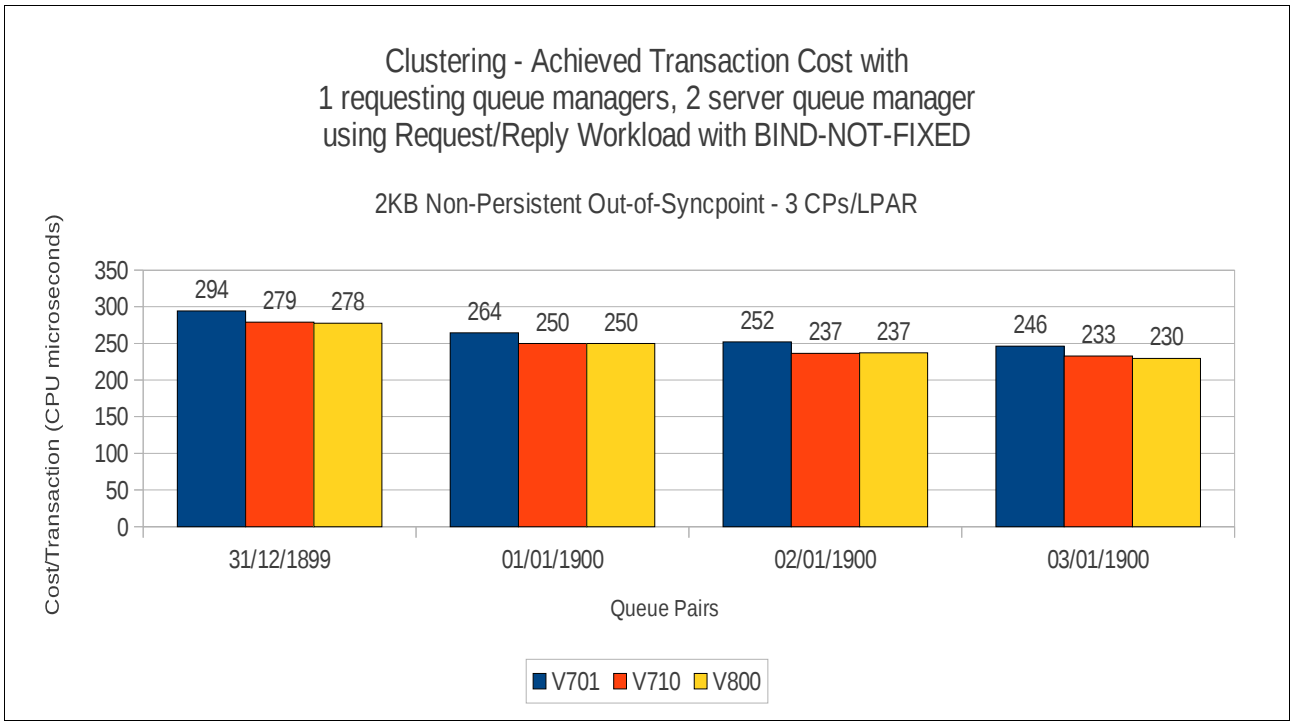
64KB messages – bind-on-open



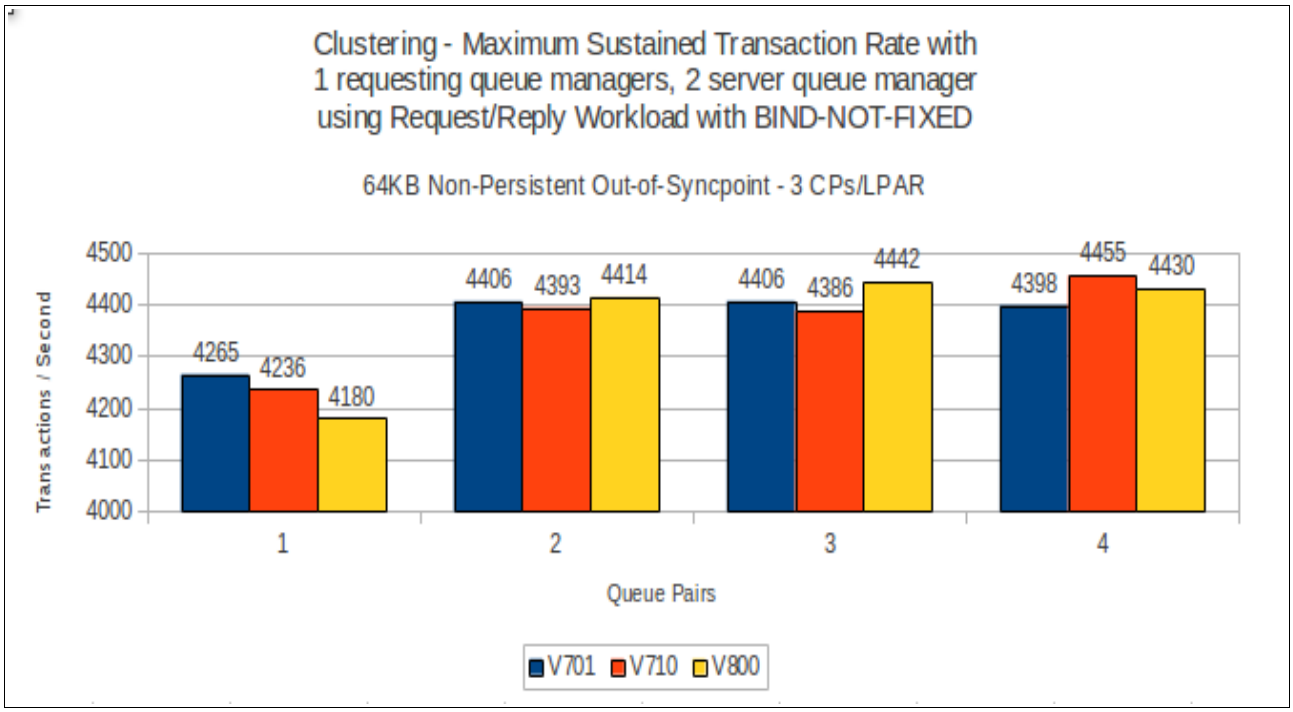


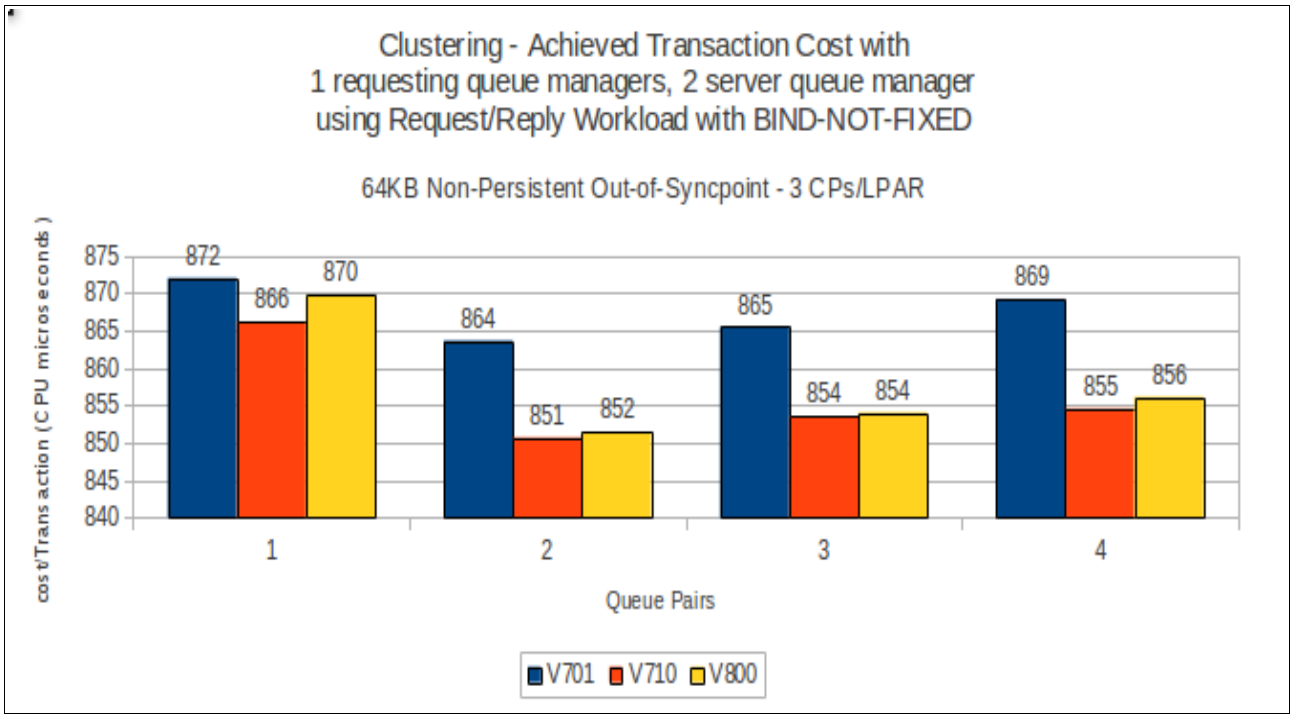
2KB messages – bind-not-fixed





64KB messages – bind-not-fixed





Regression – moving messages across SVRCONN channels

The regression tests for moving messages across SVRCONN channels have a pair of client tasks for each queue that is hosted on the z/OS queue manager.

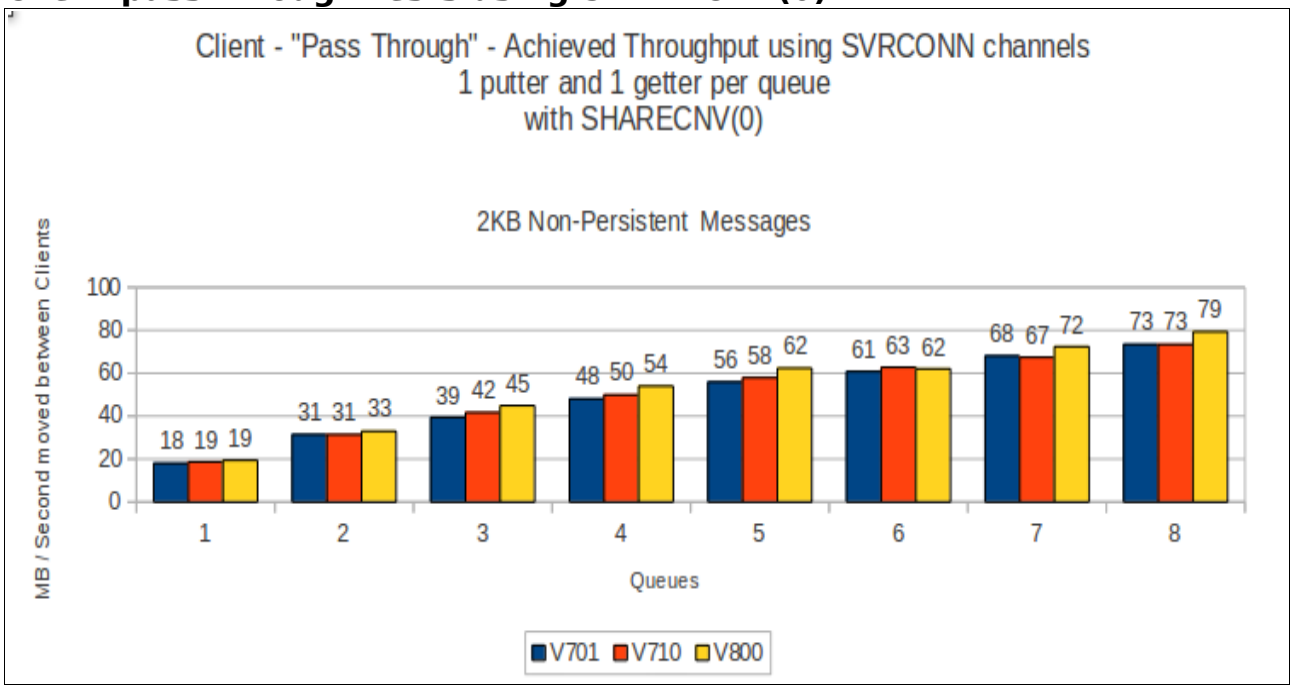
One of the pair of client tasks puts messages to the queue and the other client task gets the messages from the queue. As the test progresses, an increasing number of queues is used, with a corresponding increase in the number of putting and getting clients.

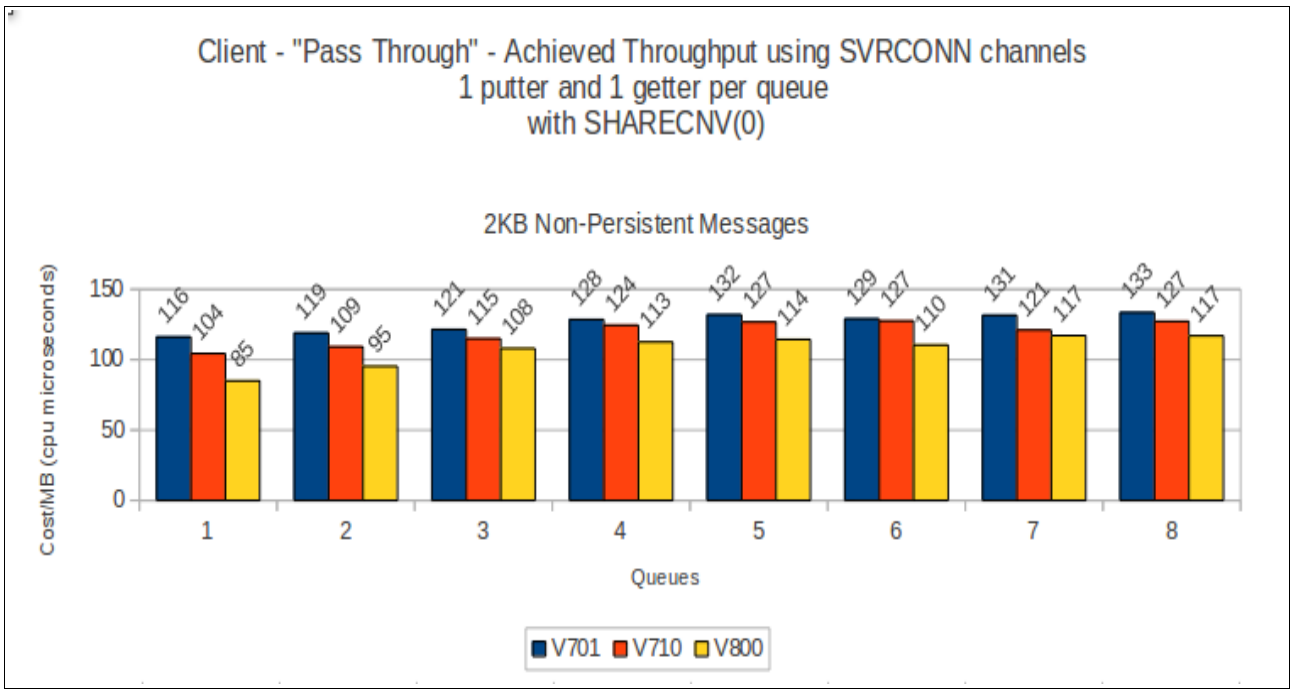
2 sets of tests are run – the first uses SHARECNV(0) on the SVRCONN channel to run in a mode comparable to that used pre-version 7. The second uses SHARECNV(1) so that function such as asynchronous puts and asynchronous gets are used via the DEFPRESP(ASYNC) and DEFREADA(YES) queue options.

Choosing which SHARECNV option is appropriate can make a difference to performance and is discussed in more detail in the supportpac MP16 – “Capacity Planning and Tuning Guide”.

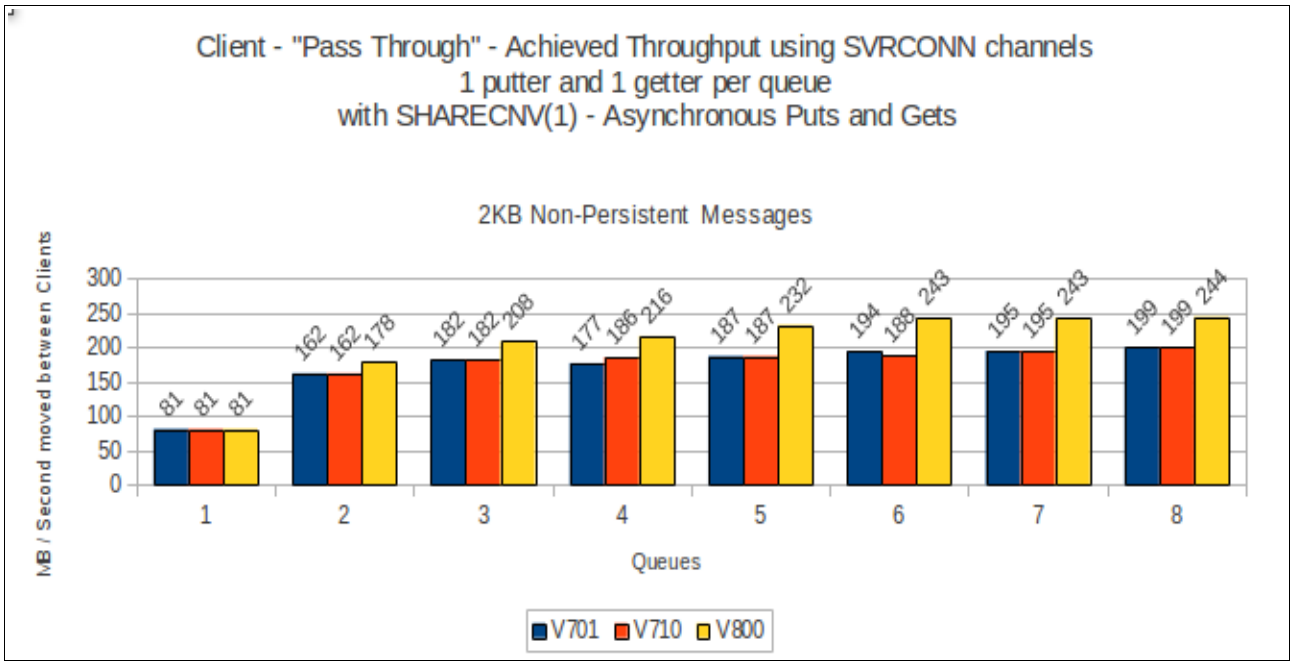
NOTE: The rate and costs are based on the number of MB of data moved per second rather than the number of messages per second.

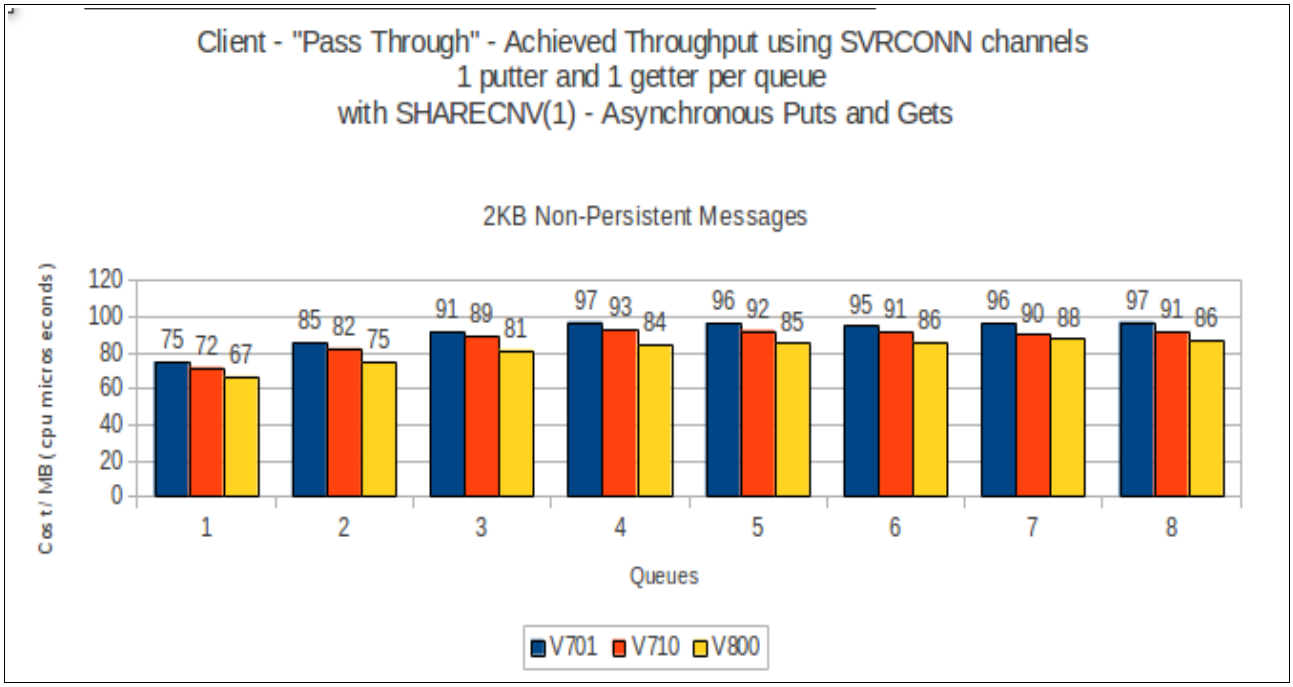
Client pass through tests using SHARECNV(0)





Client pass through tests using SHARECNV(1)





Regression – IMS bridge

The regression tests used for the IMS bridge use 3 queue managers in a queue sharing group (QSG) each on separate LPARs. A single IMS region is started on 1 image and has 16 Message Processing Regions (MPRs) started to process the MQ workload.

The IMS region has been configured as detailed in the supportpac MP16 using the recommendations in the section “IMS Bridge: Achieving Best Throughput”.

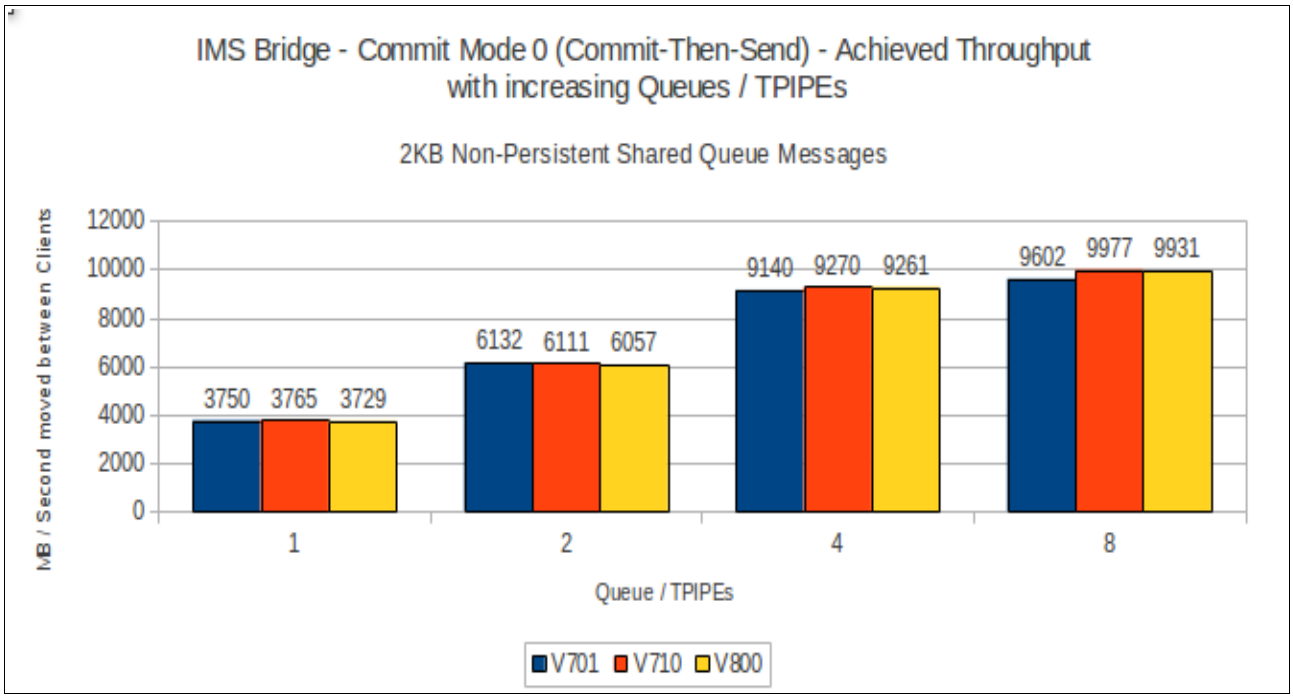
NOTE: 16 MPRs are more than really required for the 1, 2 and 4 TPIPE tests but they are available for a consistent configuration across the test suite.

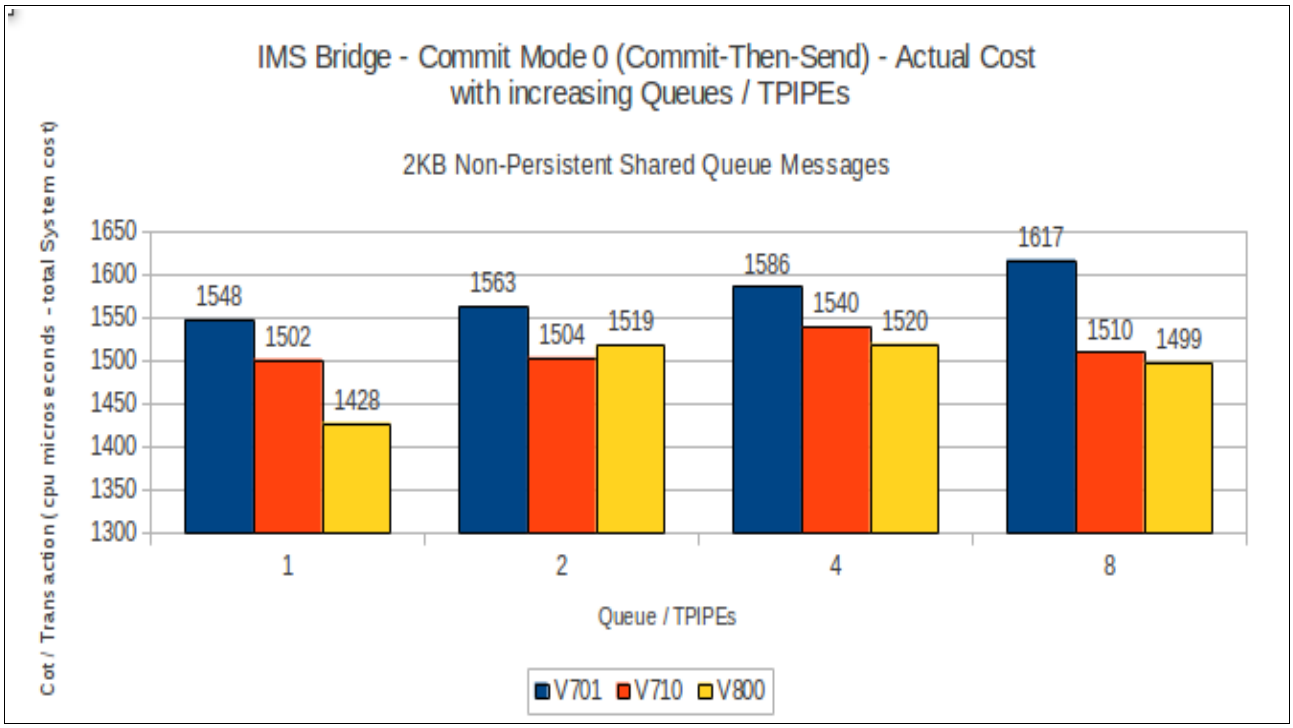
There are 8 queues defined in the QSG that are configured to be used as IMS Bridge queues.

Each queue manager runs a set of batch requester applications that put a 2KB message to one of the bridge queues and waits for a response on a corresponding shared reply queue.

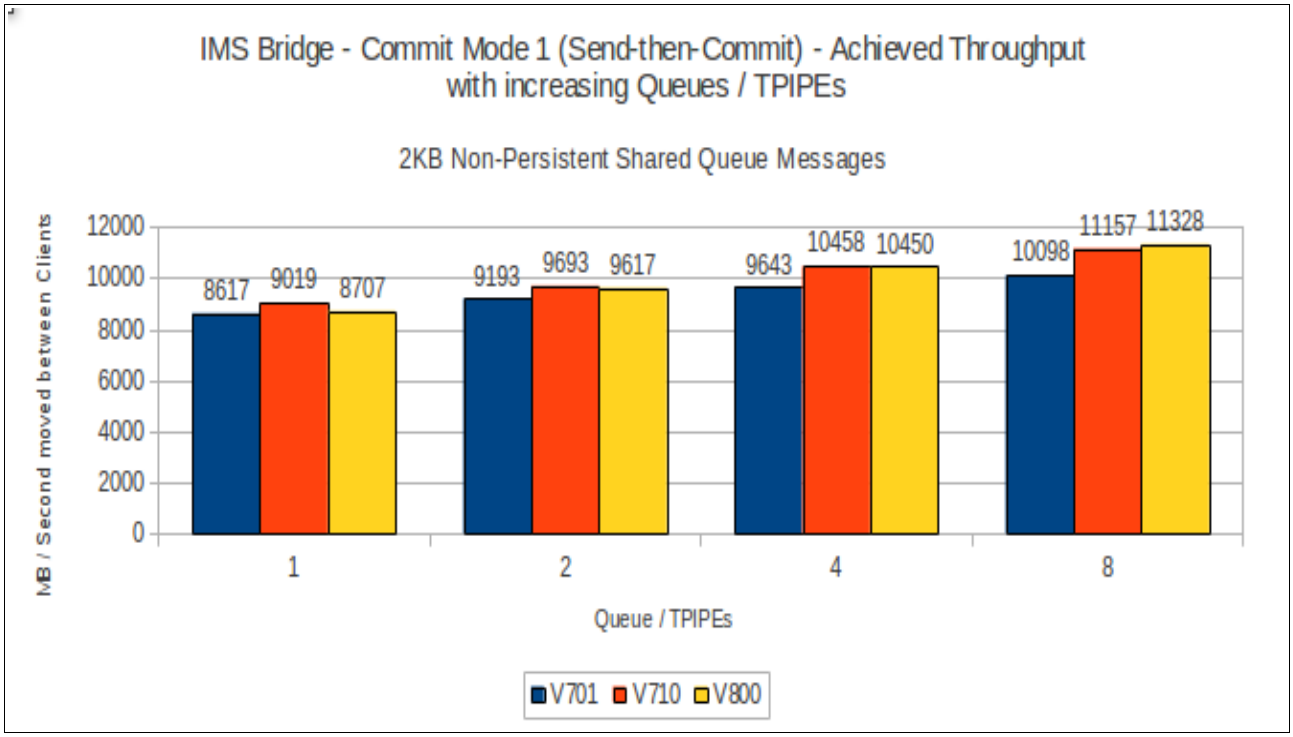
Tests are run using both Commit Mode 0 (Commit-Then-Send) and Commit Mode 1 (Send-Then-Commit)

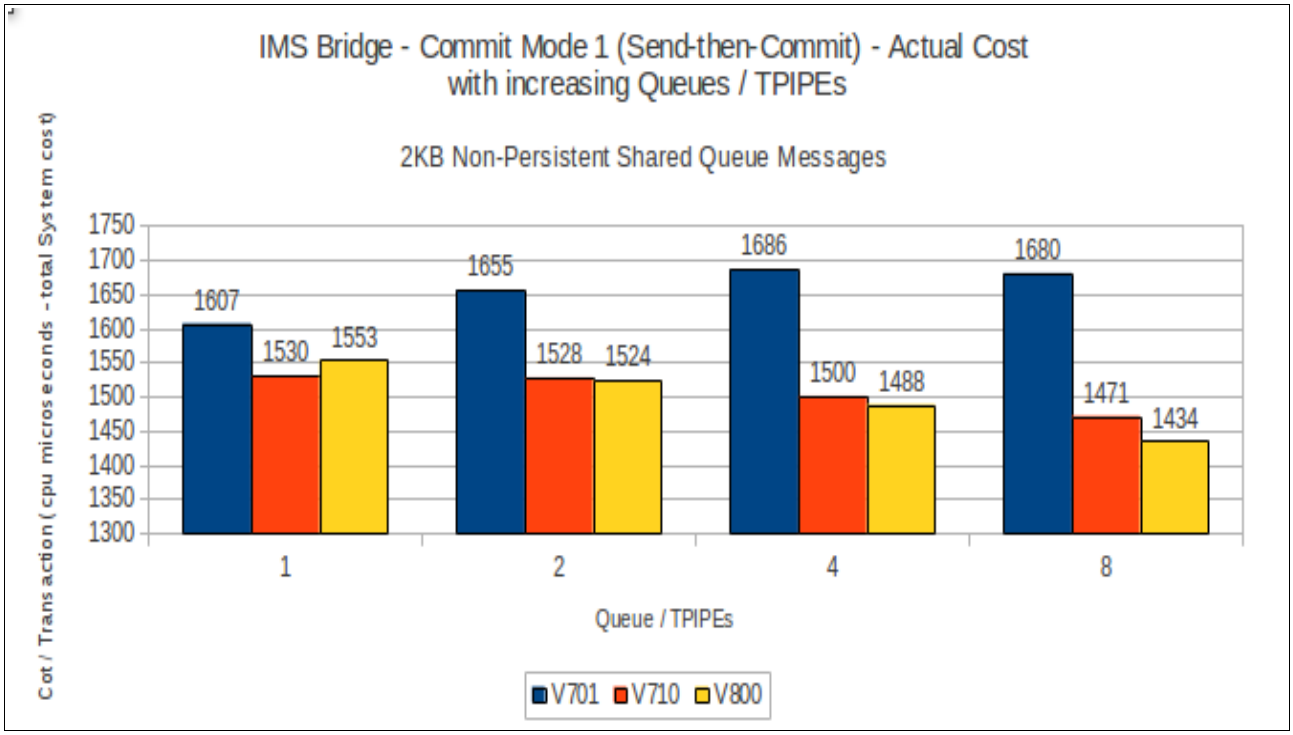
Commit mode 0 (commit-then-send)





Commit mode 1 (send-then-commit)





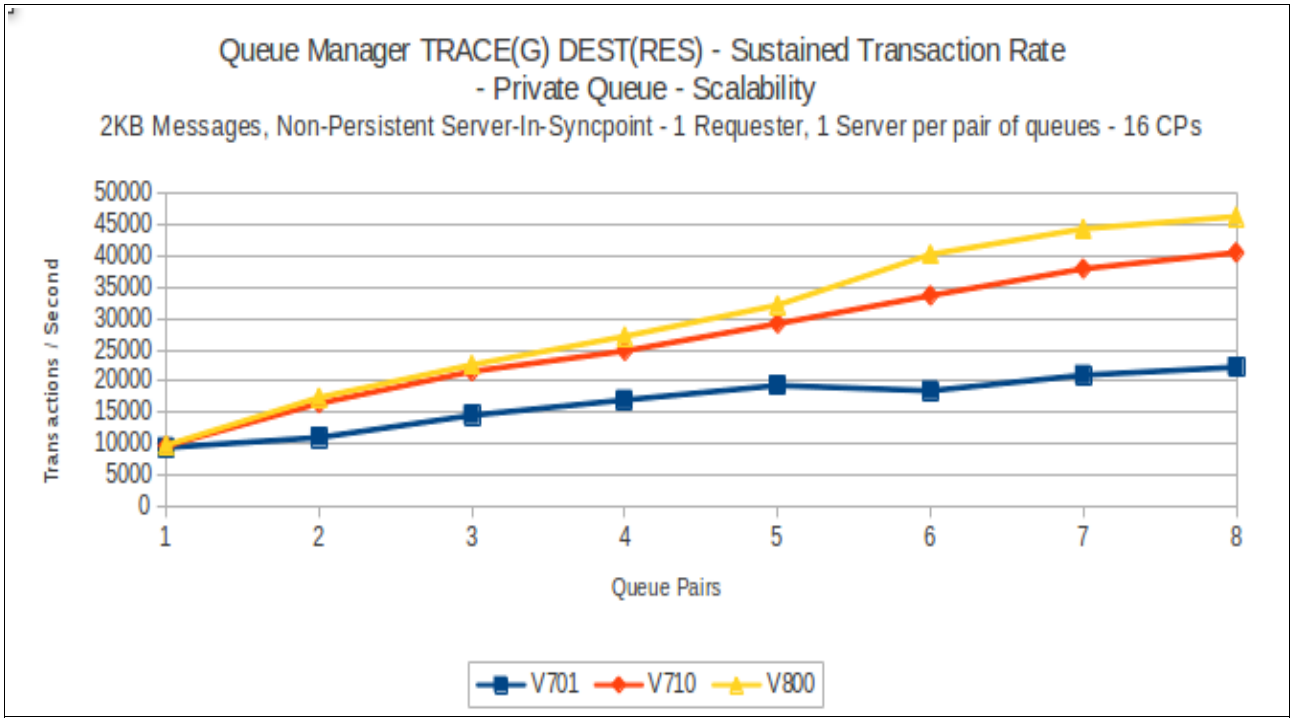
Regression – Trace

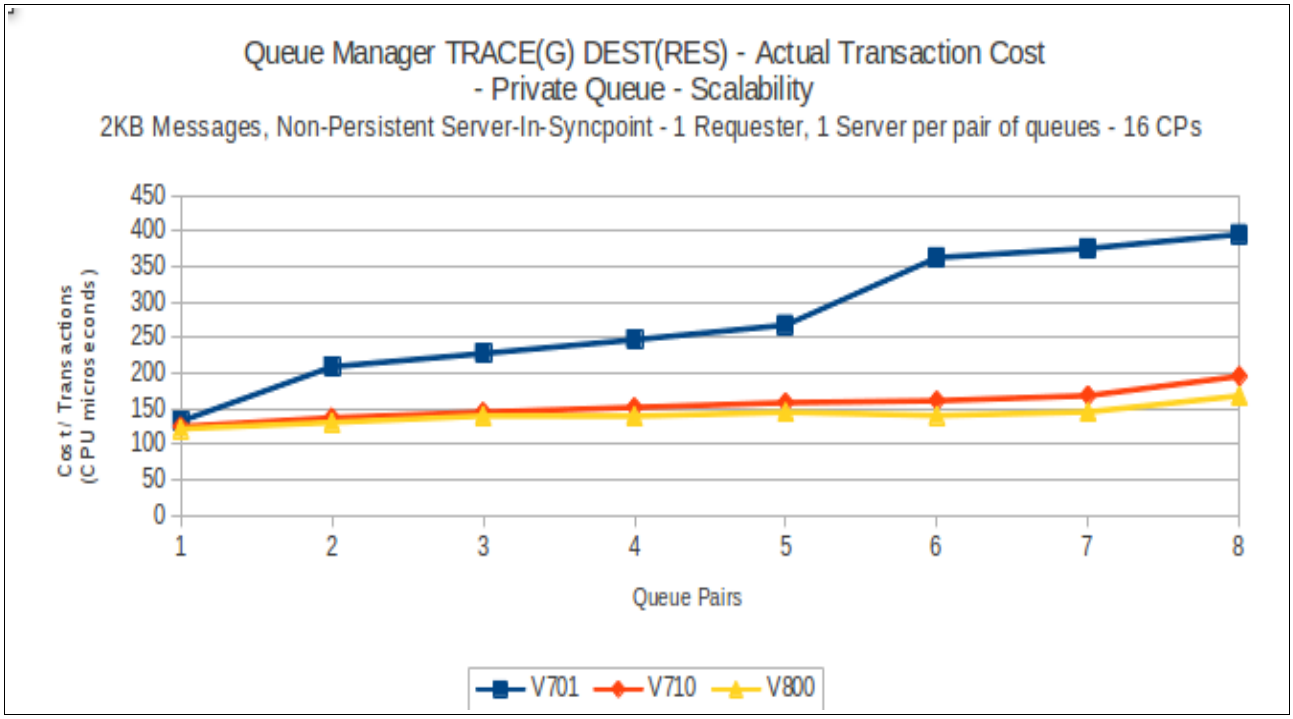
The regression tests for trace cover both queue manager global trace and the channel initiator trace.

Queue manager global trace

The queue manager global trace tests are a variation on the private queue non-persistent 2KB scalability tests with TRACE(G) DEST(RES) enabled.

From version 7.1, the global trace uses 64-bit storage which allows the queue manager to store a trace record for each thread and reduces contention on the trace storage. As a result the throughput achieved is significantly improved over previous releases of WebSphere MQ for z/OS.


Queue Manager TRACE(G) DEST(RES) - Sustained Transaction Rate
- Private Queue - Scalability
2KB Messages, Non-Persistent Server-In-Syncpoint - 1 Requester, 1 Server per pair of queues - 16 CPs
Transactions / Second
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
1
2
3
4
5
6
7
8
Queue Pairs
V701
V710
V800

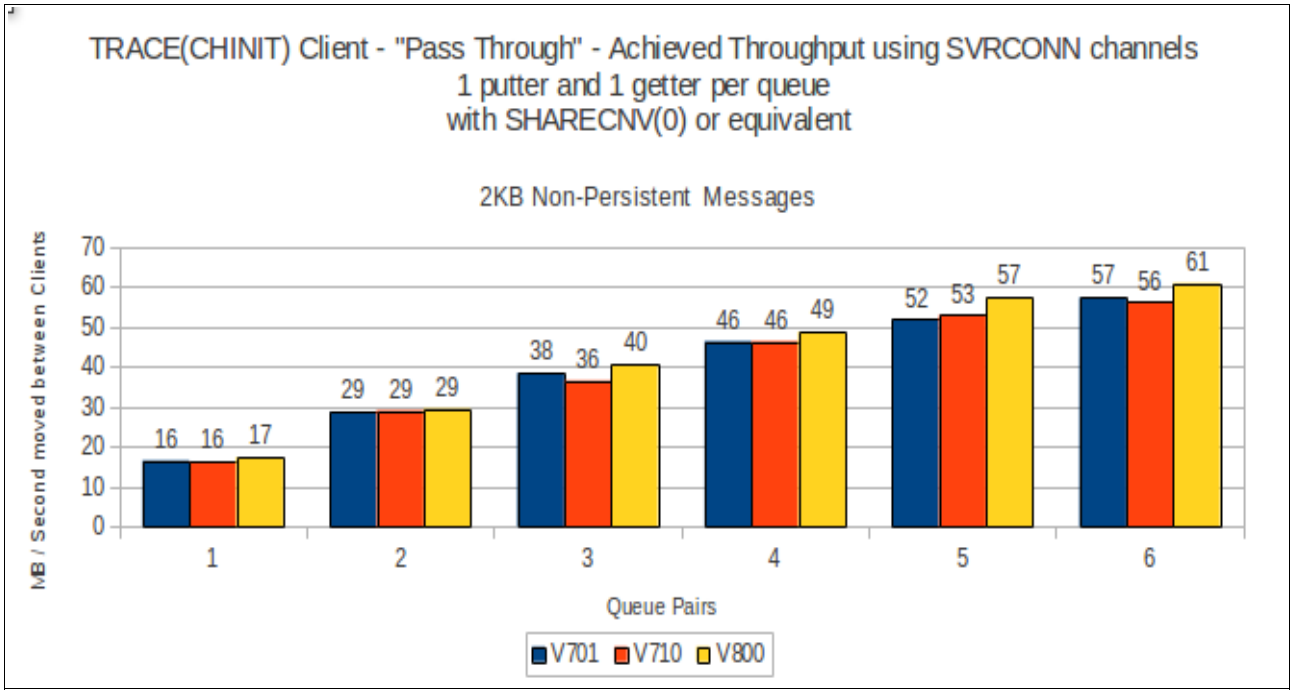

Queue Manager TRACE(G) DEST(RES) - Actual Transaction Cost
- Private Queue - Scalability
2KB Messages, Non-Persistent Server-In-Syncpoint - 1 Requester, 1 Server per pair of queues - 16 CPs
Cost/ Transactions
(CPU microseconds)
450
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
Queue Pairs
V701
V710
V800

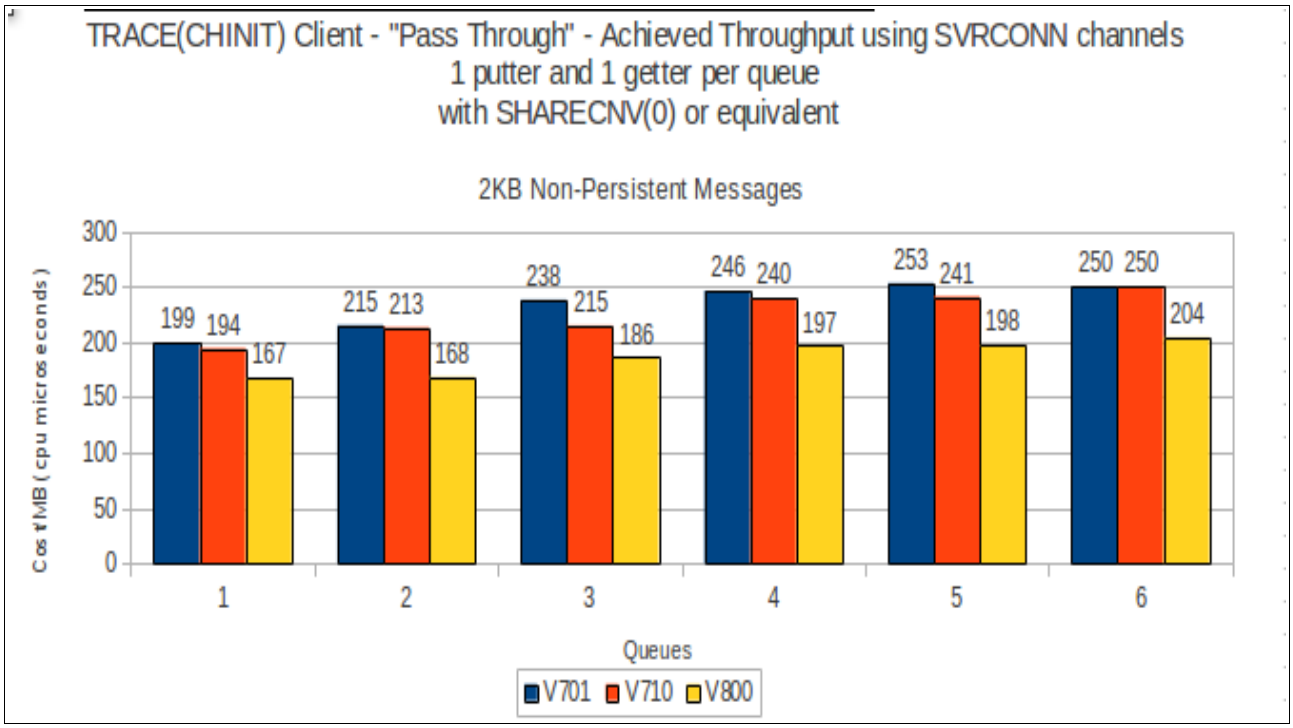
Channel initiator trace

The channel initiator trace tests are a variation on the moving messages across SVRCONN regression tests using channels with SHARECNV(0) and TRACE(CHINIT) enabled.

In V7.0.1 a significant increase in the number of trace points recorded was added, which had the effect of increasing the overhead of channel initiator trace over V6.0.0.

In V7.1.0, the trace routines are called more efficiently than in V7.0.1 but there are still more trace points being recorded than V6.0.0.


TRACE(CHINIT) Client - "Pass Through" - Achieved Throughput using SVRCONN channels
1 putter and 1 getter per queue
with SHARECNV(0) or equivalent
2KB Non-Persistent Messages
MB / Second moved between Clients
70
60
50
40
30
20
10
0
16 16 17
29 29 29
38 36 40
46 46 49
52 53 57
57 56 61
1
2
3
4
5
6
Queue Pairs
V701 V710 V800


TRACE(CHINIT) Client - "Pass Through" - Achieved Throughput using SVRCONN channels
1 putter and 1 getter per queue
with SHARECNV(0) or equivalent
2KB Non-Persistent Messages
Cost/MB (cpu microseconds)
300
250
200
150
100
50
0
199 194 167
215 213 168
238 215 186
246 240 197
253 241 198
250 250 204
1
2
3
4
5
6
Queues
V701 V710 V800

Appendix B – System configuration

MVS: zEnterprise EC12 (2827-7A1) configured thus:

LPAR 1

Between 1 and 16 dedicated CP processors

LPAR 2

Between 1 and 3 dedicated CP processors

LPAR 3

Between 1 and 10 dedicated CP processors.

Default configuration:

3 dedicated processors on each LPAR

Coupling Facility

Internal coupling facility with 4 dedicated processors.
Dynamic Dispatching switched OFF

DASD
FICON-connected DS8870
4 dedicated channel paths (shared across sysplex)
HYPERPAV enabled.

System Settings:

Each MVS image running z/OS v2r1
Coupling facility running CFCC level 19
ZHPF available, but by default is configured as disabled.
HIPERDISPATCH enabled by default.
Tests moving messages between LPARs that were on different TCP/IP subnets using a 10Gb network.

Trace Status:

- TRACE(GLOBAL) disabled.
- TRACE(S) enabled
- TRACE(A) CLASS(3) enabled

General Information

- Client machine was:
  - IBM SYSTEM X3850 - 6 x 3.16GHz Processor, 6Gb Memory
- Client tests used a 10Gb performance network

- Other IBM products used:
  - CICS V6.7 CTS4.2 with latest service applied as of May 2014.
  - IMS v12
  - WebSphere MQ v7.0.1 with latest service applied as of May 2014.
  - WebSphere MQ v7.1.0 with latest service applied as of May 2014.