Performance Report - SupportPac MP1G
WebSphere MQ for z/OS V7.0.1

Version 1.0

October 2009

Tony Sharkey
Pete Hickson

Take Note!

Before using this report please read the general information under "Notices"

First Edition, October 2009
This edition applies to Version 7.0.1 of WebSphere MQ for z/OS.

Notices

This report is intended for Architects, Systems Programmers, Analysts and Programmers wanting to understand the performance characteristics of **WebSphere MQ for z/OS V7.0.1**. 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.

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.

Information contained in this report has not been submitted to any formal IBM test and is distributed “as is”. The use of this information and the implementation of any of the techniques is the responsibility of the customer. Much depends on the ability of the customer to evaluate these data and project the results to their operational environment.

The performance data contained in this report was measured in a controlled environment and results obtained in other environments may vary significantly.

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:

Enterprise Storage Server
IBM
SupportPac
WebSphere
WebSphere MQ
z/OS
zSeries

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

Summary of Amendments

Date	Changes
October 2009	Initial Version

Table of Contents

Performance Highlights – WebSphere MQ for z/OS v7.0.1

This report focuses on performance changes since the previous versions (V6.0.0 and V7.0.0) and on the performance of new function in this release.

SupportPac MP16 "Capacity Planning and Tuning For WebSphere MQ for z/OS" will be updated to include WMQ Version 7.0.1 information. MP16 will continue to be the repository for ongoing advice and guidance learnt 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 7.0.1 for typical messaging workloads.

Page Set Usage with Small Messages

As processors become faster and using more processors becomes more common, it becomes more important that the MQ queue manager operates in an optimum manner.

A queue with relatively few[1] messages can hold all of its messages in the associated bufferpool. This means the message can be put and gotten much faster than if the message has been written to the page set.

In a high-workload, low queue depth environment, it is necessary for the messages to remain in buffers to ensure the best performance is obtained.

MQ operates a scavenger process to remove "dead" pages, allowing them to be re-used.

For messages that fit on one or more 4K pages, the scavenger process can be run immediately after the message has been deleted, which leave the number of used pages to be an accurate measure of the number of messages on the queue.

Prior to version 7.0.1, small messages were stored such that multiple messages could co-exist on the same page. This meant that the scavenger could not run once a message was deleted as there were potentially other messages still on the page. Instead, the small message scavenger would run periodically – up to every 5 seconds . This means that a workload using small messages could see a build up of dead pages that were waiting to be scavenged. With ever faster processors, the time taken to fill the bufferpool with dead pages becomes significantly reduced. In turn, this meant that the messages would overflow onto the pageset and potentially the queue manager could be spending time performing I/O – causing slower MQGETs and MQPUTs.

[1] The number of messages that can be held in the bufferpool depends on the size of the bufferpool and the size and number of messages on the queue.

To allow the scavenger to work more efficiently with small messages, each message is now stored in a separate page. In addition a separate index page is used to hold data for approximately 72 messages. Once the message is deleted, the page holding the message data can be re-used immediately but the index page can only be scavenged once all messages referenced are deleted.

We recognise that not all customers operate in a high workload, low queue depth environment and the increase in storage usage for small messages may affect other factors in the queue managers' performance. For example, it will take less small messages to cause pageset expansion than previously.

To this end, it is now possible to force the queue manager to store multiple small messages on a page as per releases prior to version 7.0.1 by adding the following statement to the CSQINP2 DD card:

REC QMGR (**TUNE MAXSHORTMSGS 0**)

Using this "maxshortmsgs" tuning attribute means that the small message scavenger uses pre-version 7.0.1 function.

We do not recommend overriding the default behaviour of the queue manager except under the direction of IBM service.

The benefits of the new scavenger process can be seen in the section "Small Message Scavenger"

The following table shows the size of page sets for a given number of messages:

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

Above Bar Storage Usage

Since z/OS V1.2, 64-bit virtual storage has been available within a single address space. This 64-bit storage is also known as "above the bar" storage. The "bar" refers to the 2 GB line that was the 31-bit virtual addressing limit. The introduction of z/OS V1.5 allowed for shared 64-bit virtual storage.

WebSphere MQ V7.0 is the first MQ release on z/OS to begin to exploit this feature. Version 7.0.1 extends the usage of 64-bit storage.

In version 7.0.1, two further areas of the queue manager have been changed to exploit the 64-bit "above the bar" storage.

- Indexed queues
- Security manager

Using 64-bit storage to hold data for queue indices, intra-group queuing and publish/subscribe allows the queue manager to use the below the bar storage for other purposes. It should be noted that usage of the 64 bit storage can require an increase to amount of space available to the MVS paging datasets.

This constraint relief to indexed queues and security manager means that the storage previously used to hold index data and security data inside the queue managers address space is now available for other purposes.

Indexed Queues

WebSphere MQ for z/OS version 7.0.1 moves the storage used for the index for indexed queues into above the bar storage.

Rather than being restricted to approximately 1.6GB minus queue manager usage, the index is no longer as constrained at it has access to all of the storage above the bar.

Security Manager

WebSphere MQ for z/OS version 7.0.1 moves the storage used for security management into above the bar storage.

This constraint relief means that the storage previously used to hold security data inside the queue managers address space is available for other purposes.

When the security timeout period is reached, the storage obtained by security manager is released back to a pool for re-use, rather than being released.

This re-using of the pooled memory uses less CPU cycles than the initial allocation.

There is no difference in ECSA usage in a version 7.0.1 queue manager whether using 1 or 50,000 unique user ids.

New Function

Log Compression

WebSphere MQ for z/OS version 7.0.1 introduces a new option to enable the compression of log data. This can be enabled by issuing

```
SET LOG COMPLOG(RLE)
```

The log compression process applies RLE (Run Length Encoding) compression.

Log compression is either enabled for all queue manager work or none.

RLE compression is ideal for message data which includes large numbers of repeating characters; a printed report that includes long strings of blanks, for example, might be highly compressible using the RLE technique.

Compression of log data can reduce the amount of data being written to the active logs and can therefore improve the throughput of persistent message workload.

Even with totally incompressible messages, MQ can compress its own MQ headers, making a saving on the amount of data that is logged.

The CPU costs of log compression vary with the nature of the message data and are not readily predictable.

Log data is written in chunks of 4KB. The log compression routine attempts to compress data in those 4KB chunks, so a 100KB message will be split into 25 chunks and each chunk will be compressed, rather than compressing the 100KB message in a single attempt.

On a system where the DASD logging rate is the constraining factor, log compression can help improve throughput.

Log compression does not come at zero cost.

In a CPU constrained system, the transaction rate may be degraded by using log compression

The benefits to be gained from log compression depend significantly on the actual message payload. If the data does not contain repeating characters, log compression may not be the answer.

Select on Get

WebSphere MQ for z/OS version 7.0.1 introduces the capability to specify a message selection criteria at the time the MQOPEN (for input) is issued.

This allows messages to be selected from a queue where the messages on the queue or messages to be put to the queue contain message properties[2].

By specifying an MQOD "SelectionString", it is possible to notify the queue manager of the message selection criteria in order to limit the messages provided to the application to those actually required by the application.

In a client/server environment, this has an additional benefit. In previous releases of MQ, all messages would flow to the MQ client, only for the MQ code to filter out the unwanted messages. In version 7.0.1, only the selected messages are sent over the network to the client. This filtering by the queue manager should reduce the load on the network.

[2] The correlation ID may be specified, so even if a message does not contain properties, selection criteria can still be applied.

Performance Data

Small Message Scavenger

The changes to the way small local messages are stored have improved the scalability of using small messages on local queues and may improve how long pages are used in the queue at a cost of slightly higher CPU, and more total pages.

1. Increase the amount of storage required to store small messages.. By changing the way messages fewer than 1568 bytes are stored so that each message uses a 4KB page, less messages can be stored on a pageset for a V701 queue manager than a V600 or indeed a V700 queue manager. In the event of a system outage that causes messages to build up on a queue, it will require fewer messages to cause pageset expansion and also to reach the point where the putting application is returned an MQ return code 2192 "MQRC_STORAGE_MEDIUM_FULL"

2. Reduce the time taken to scavenge dead pages. By reducing the time that the scavenger takes to complete its work, the sooner those dead pages are available to be re-used.

    - MQ attempts to keep the messages in buffers for as long as possible as it is much faster to access memory than to access DASD.
    - If there is not enough space in the buffer, the data needs to be written to the pageset DASD.
    - When a pageset reaches its 85% used threshold, the rate of messages put is slowed to allow the pageset to expand.

    By reducing the time the scavenger runs, it is less likely that in a high-workload low-queue depth workload that the buffers will become full, causing overflow into pagesets and also less likely that pagesets will get expanded.

The benefits of the changes to small messages and the scavenger can be seen in the scenarios described below.

Out-Of-Syncpoint Workload

This section measures show the benefits of the enhanced small message scavenger in 2 scenarios where all messages are put and gotten with no syncpoint.

1. Low contention on Queues – find the maximum sustainable transaction rate when the workload is spread across multiple queues.
2. High contention on Queues – find the maximum sustainable transaction rate using a single pair of queues.

Low Contention on Queues with all messages out of syncpoint

The following chart shows the transaction rate achieved on a 17-processor LPAR of a z10 EC64. In this measurement, a transaction is defined as:

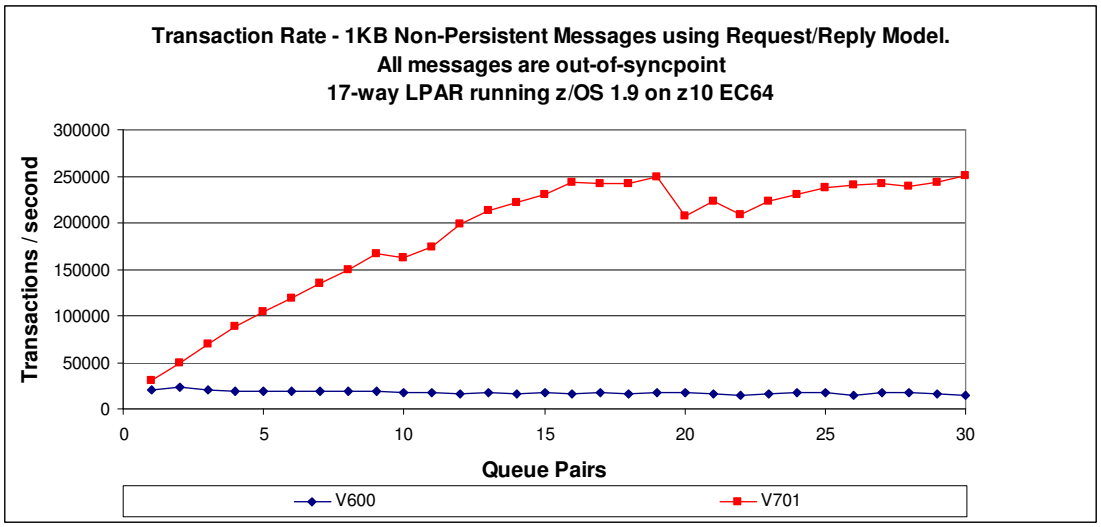
- Requester application puts a message to a queue and goes into MQGET-with-wait for reply on separate queue that is indexed by correlation ID.
- Server application gets a message from the queue and puts the reply to the reply-to queue
- Requester application gets the message using a known correlation ID.
- All work is performed out of syncpoint.

Initially there was 1 batch requester task and 1 batch server task per request/reply queue pair. As the test progressed, more pairs of queues were used – from 1 to 30.

There should never be more than 1 message on any pair of queues.

The queue manager has been configured as follows:

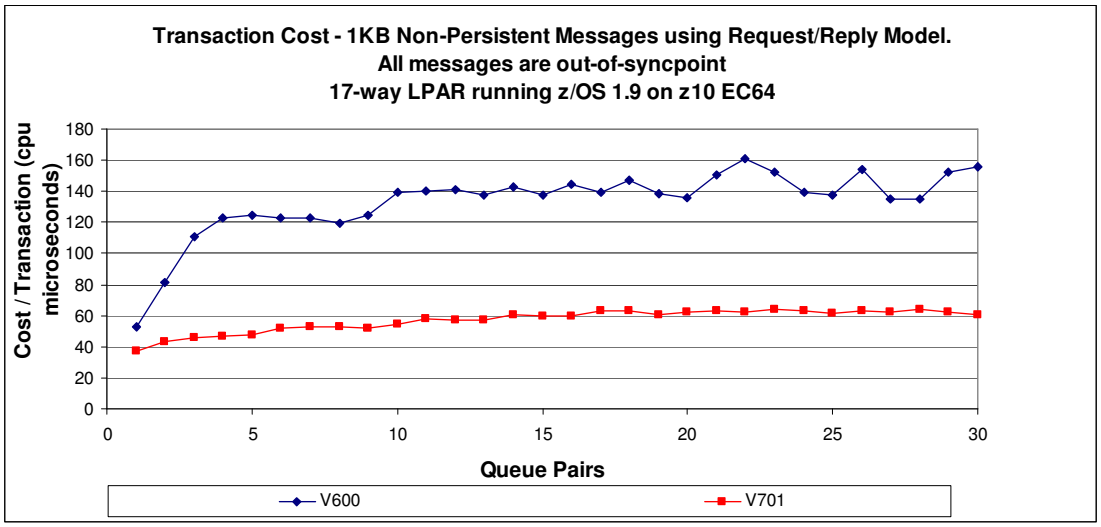
- 16 buffer pools (0 to 15) each of size 10,000 buffers.
- 16 pagesets (0 to 15), with pagesets 1 to 15 expanded to 4GB.
- 30 request queues spread evenly over pagesets 1 to 15, i.e. 2 queues per pageset.
- 30 reply queues, spread evenly over pagesets 1 to 15 and are located on the same pageset as their corresponding request queue.
  All reply queues have INDXTYPE(CORRELID).



The above chart shows a much smoother increase in the transaction rate as more queues are used than was achievable in previous releases of MQ as well as a higher peak transaction rate – 250,000 transactions per second.

Note the rate drop at 20-22 queue pairs for V7.0.1 – This is a point where the queue manager was restarted and there was a period where the transaction rate took time to stabilise.

The transaction rate peaks at 15 queue pairs. Once the measurement goes beyond 15 queue pairs, an increasing number pagesets and therefore buffer pools in use are shared. CPU also becomes a constraining factor.



The previous chart shows the cost per transaction as more queue pairs are used.

The costs are based on the total amount of CPU used by the queue manager, requester and server tasks divided by the number of transactions per second.

Reviewing the data generated by the queue manager running with TRACE(A) CLASS(3) shows that the MQGET CPU cost in this measurement for version 6.0 is 24 microseconds compared to 10 microseconds in version 7.0.1.

High Contention on Queues with all messages out of syncpoint

The following measurement compares the throughput achieved on a single queue pair for a range of message sizes when using a version 7.0.0 queue manager and a version 7.0.1 queue manager when multiple requester and server tasks are used.

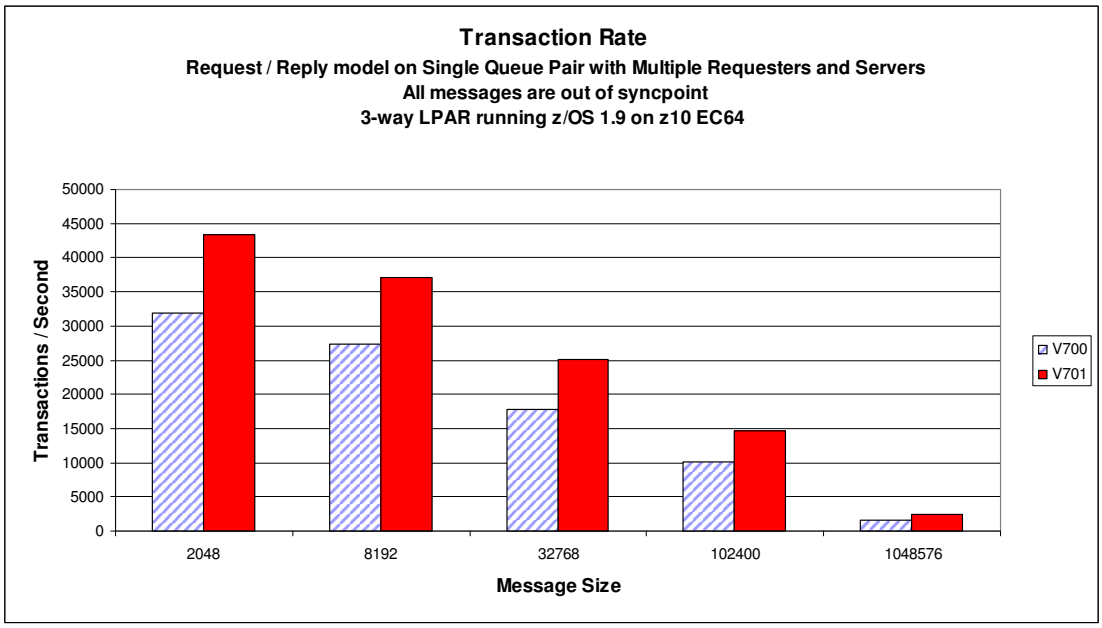
In these measurements, the following configuration is used:

- A single z/OS 1.9 LPAR with 3 dedicated processors on a z10 EC64
- 5 batch requester applications put a message to a request queue. All requester tasks then go into an MQGET-with-wait on an indexed reply queue.
- 4 batch server applications are running in an MQGET on the request queue. Upon successful MQGET, the applications MQPUT a reply message to the known reply queue.
- The requester application then MQGETs the reply message using a known correlation ID.
- All messages are out of syncpoint.
- A transaction is defined as each requester putting a request message and getting a reply message.

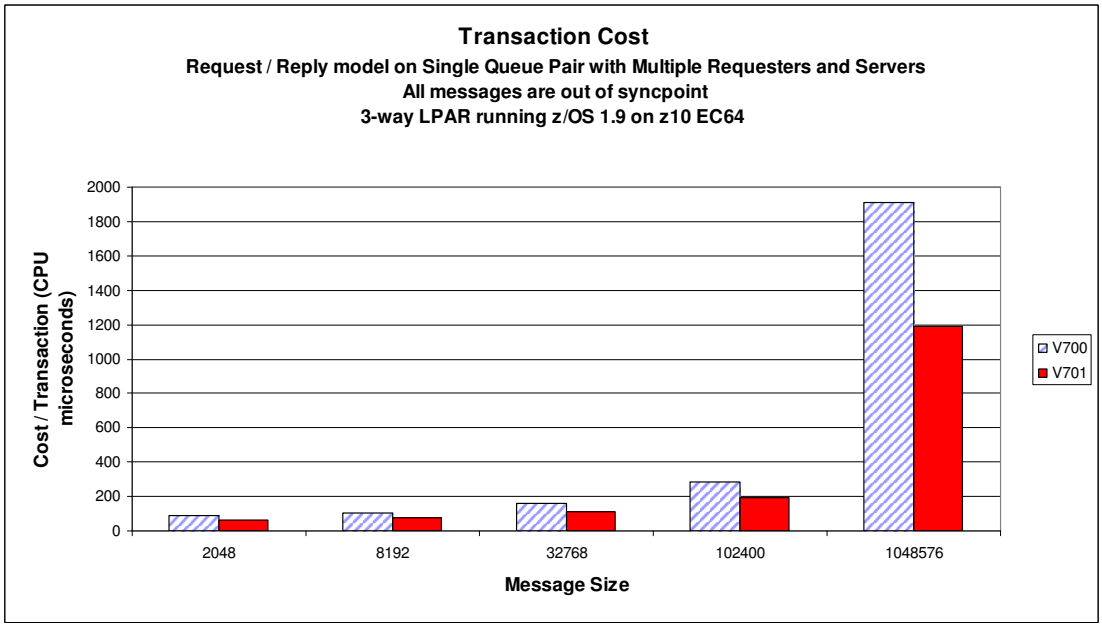
The queue manager has been configured as follows:

- 4 buffer pools (0 to 3), buffer pool 0 to 2 are defined with 20,000 buffers and buffer pool 3 has 99,000 buffers.
- 5 pagesets (0 to 4), pageset 0, 1, 2 and 4 defined with 20,000 records. Pageset 3 defined with 99,000 records.
- Request and reply queues are defined to pageset 3. All reply queues have INDXTYPE(CORRELID).
- Pageset 3 is defined to buffer pool 3.

The following chart shows the achieved transaction rate:



The following chart shows the cost per transaction for the achieved transaction rate:



The cost is calculated from the total cost of the queue manager as well as the requester and server batch applications divided by the transaction rate.

Workload where Server application gets and puts within Syncpoint

This section measures show the benefits of the enhanced small message scavenger in 2 scenarios where all messages processed by the server applications are put and gotten within syncpoint.

1. Low contention on Queues – find the maximum sustainable transaction rate when the workload is spread across multiple queues.
2. High contention on Queues – find the maximum sustainable transaction rate using a single pair of queues.

Low Contention on Queues where Server application uses Syncpoint

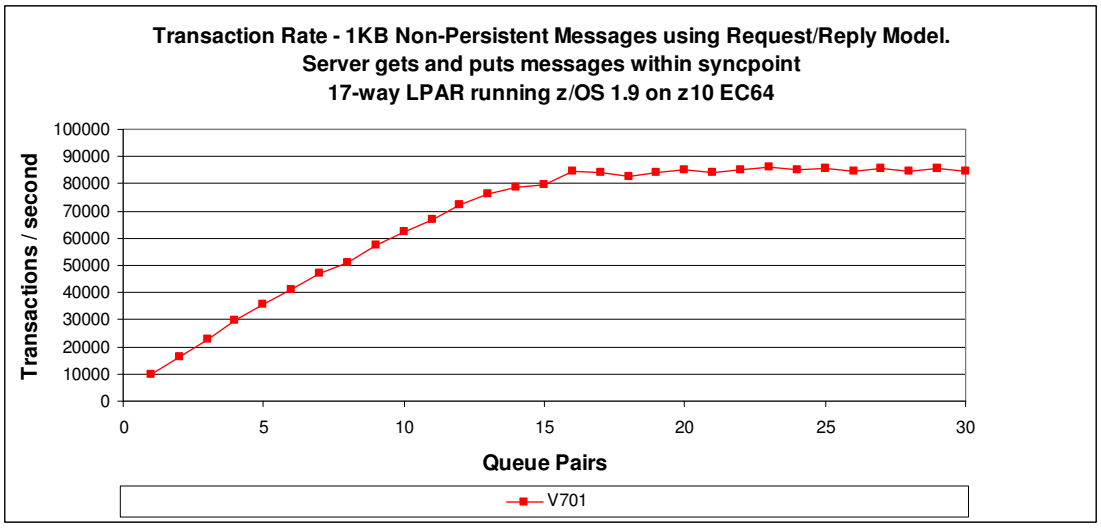
The following chart shows the transaction rate achieved on a 17-processor LPAR of a z10 EC64. In this measurement, a transaction is defined as:

- Requester application puts a message to a queue and goes into MQGET-with-wait for reply on separate queue that is indexed by correlation ID.
- Server application gets a message from the queue and puts the reply to the reply-to queue. **The MQGET and MQPUT are within syncpoint.**
- Requester application gets the message using a known correlation ID.

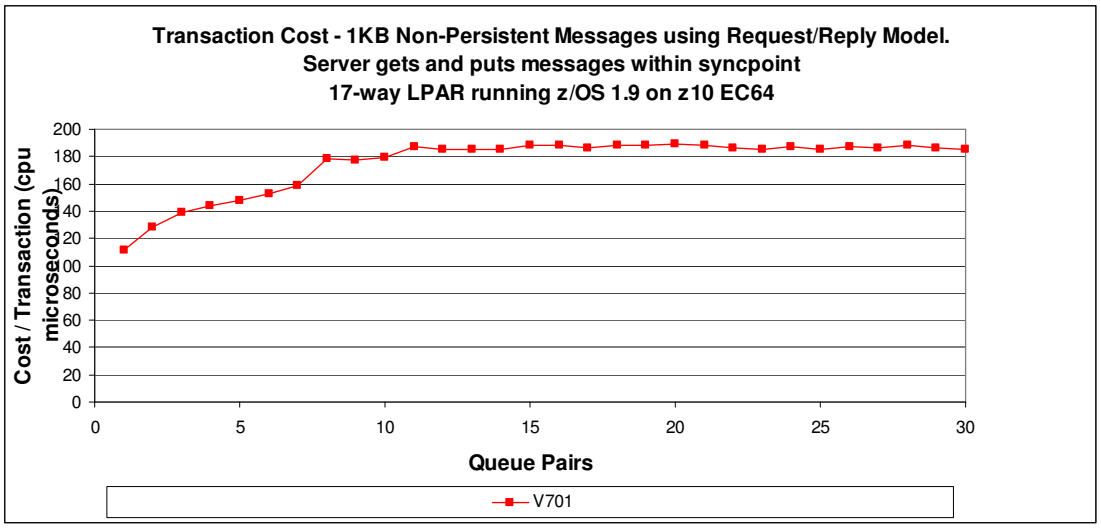
Initially there is 1 batch requester task and 1 batch server task per request/reply queue pair. As the test progresses, more pairs of queues are used – from 1 to 30.

There should never be more than 1 message on any pair of queues.

The first chart shows the transaction rate achieved in this environment. The transaction rate achieved rises steadily until the number of queue pairs in use corresponds with the number of processors available, i.e. 17 and then levels out.



The second chart shows the transaction cost which rises steadily up to 11 queue pairs and then levels out.



High Contention on Queues where Server application uses Syncpoint

The following measurement compares the throughput achieved on a single queue pair for a range of message sizes when using a version 7.0.0 queue manager and a version 7.0.1 queue manager when multiple requester and server tasks are used.

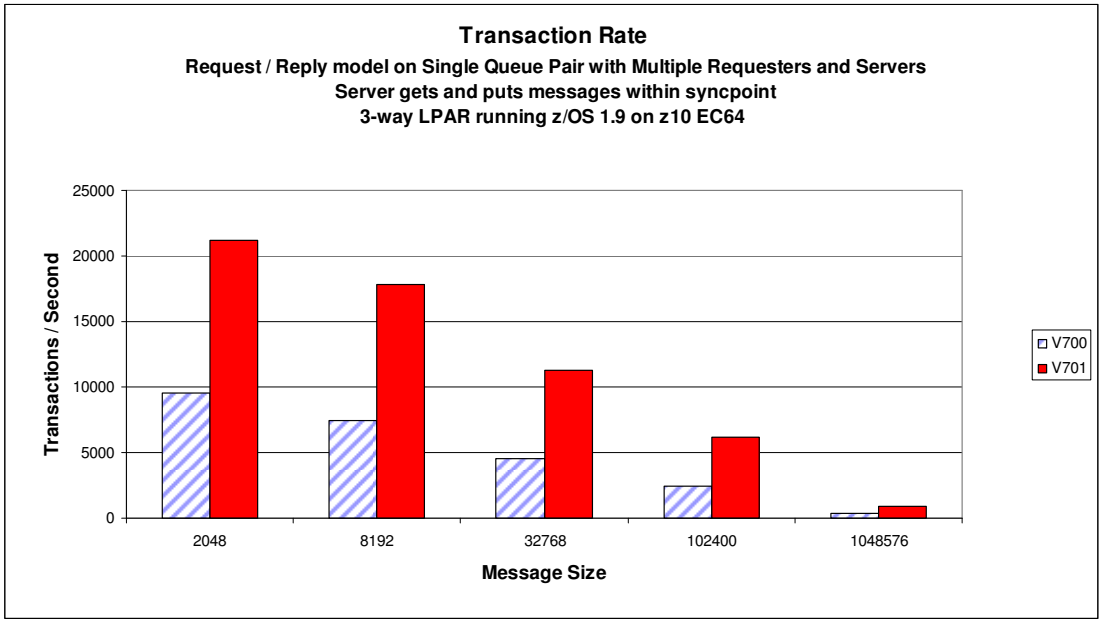
In these measurements, the following configuration is used:

- A single z/OS 1.9 LPAR with 3 dedicated processors on a z10 EC64
- 5 batch requester applications put a message to a request queue. All requester tasks then go into an MQGET-with-wait on an indexed reply queue.
- 4 batch server applications are running in an MQGET on the request queue. Upon successful MQGET, the applications MQPUT a reply message to the known reply queue. **The MQGET and MQPUT are in-syncpoint.**
- The requester application then MQGETs the reply message using a known correlation ID.
- A transaction is defined as each requester putting a request message and getting a reply message.

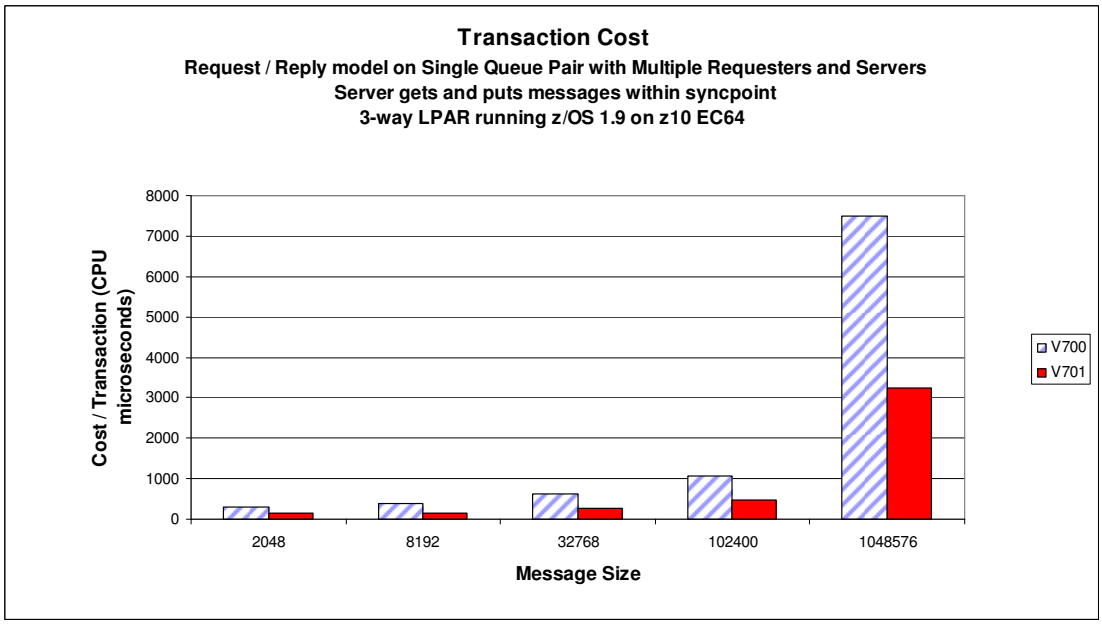
The queue manager has been configured as follows:

- 4 buffer pools (0 to 3), buffer pool 0 to 2 are defined with 20,000 buffers and buffer pool 3 has 99,000 buffers.
- 5 pagesets (0 to 4), pageset 0, 1, 2 and 4 defined with 20,000 records. Pageset 3 defined with 99,000 records.
- Request and reply queues are defined to pageset 3. All reply queues have INDXTYPE(CORRELID).
- Pageset 3 is defined to buffer pool 3.

The following chart shows the achieved transaction rate:



The following chart shows the cost per transaction for the achieved transaction rate:



Indexed Queues

There are 4 areas relating to observations made when utilising indexed queues with 64-bit indices:

1. How deep can I make my local queues?
2. Cost of putting messages to indexed queues
3. Cost of getting messages from indexed queues
4. How long will it take to restart a queue manager with deep indexed local queues?

The following section attempts to answer these questions.

How deep can I make my local queues?

As previously mentioned, the storage used to hold index data for queues has been moved into 64-bit storage.

In versions 7.0 and earlier of WebSphere MQ for z/OS, the maximum depth of an indexed queue could be limited by the amount of 31-bit storage available to the queue manager.

On our test system, this occurred at approximately **7.2 million** messages on a single queue before the queue manager reported being critically short on storage.

Using a version 7.0.1 queue manager, we were able to put in excess of **100 million**[3] messages to an indexed queue.

- In this particular case, we were limited by the size of the pageset.
- More messages could have been put, had the pageset been defined with EXPAND(USER | SYSTEM) and place in a storage class that allowed multi-volume expansion.

Putting many millions of messages to an indexed queue will use a significant amount of 64-bit storage.

- Each message put to an indexed queue uses 272 bytes of above the bar storage.

This means that for 100 million messages

- 29,940 MB (29.2GB) of storage will be used above the bar.

How does this affect you?

- The queue manager may be limited to the virtual storage it can use if an IEFUSI exit is used
  - MEMLIMIT=NOLIMIT can be coded in the queue manager proc member to override this.
- Real storage usage will increase as the number of virtual storage pages increases, particularly if the WLM storage protect option is set – so the storage is not released.
- Virtual storage usage will increase if more messages are put to indexed queues.

[3] The queue manager was configured with "**MAXSHORTMSGS 0**" to allow multiple messages per 4KB page. By default a small message in V701 will use a single 4KB page – which means a 64GB pageset can hold a maximum of 16.7 million messages.

- The following message is issued when MVS detects that 70% of the available slots are in use:

  IRA200E AUXILIARY STORAGE SHORTAGE
- To avoid this situation occurring, it may be necessary to add extra paging volumes.

The MVS command "D ASM" can show the paging volumes and how full they are, but it is rare to see the datasets above 70% before being classed as full.

Cost of putting messages to local indexed queues

In releases of MQ prior to version 7.0.1, the cost of putting messages to an indexed queue increased as the depth of the queue increased.

By using 64 bit storage to hold the index for an indexed queue, version 7.0.1 is able to maintain the cost of putting messages to an indexed queue to be the same value whether it is the 1st or the 100,000,000th message put to the queue.

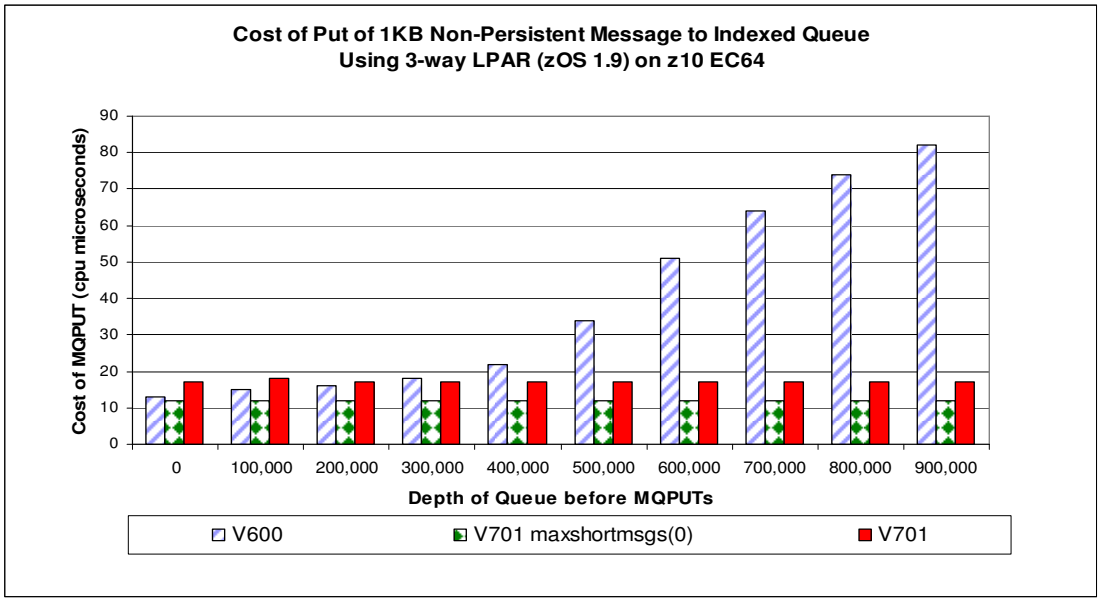
The chart below shows the cost for a batch application putting 1KB non-persistent messages to an indexed queue, when the target queue already holds an increasing number of messages.

The queue manager is configured such that the indexed queue is held on a buffer pool with 200,000 records.

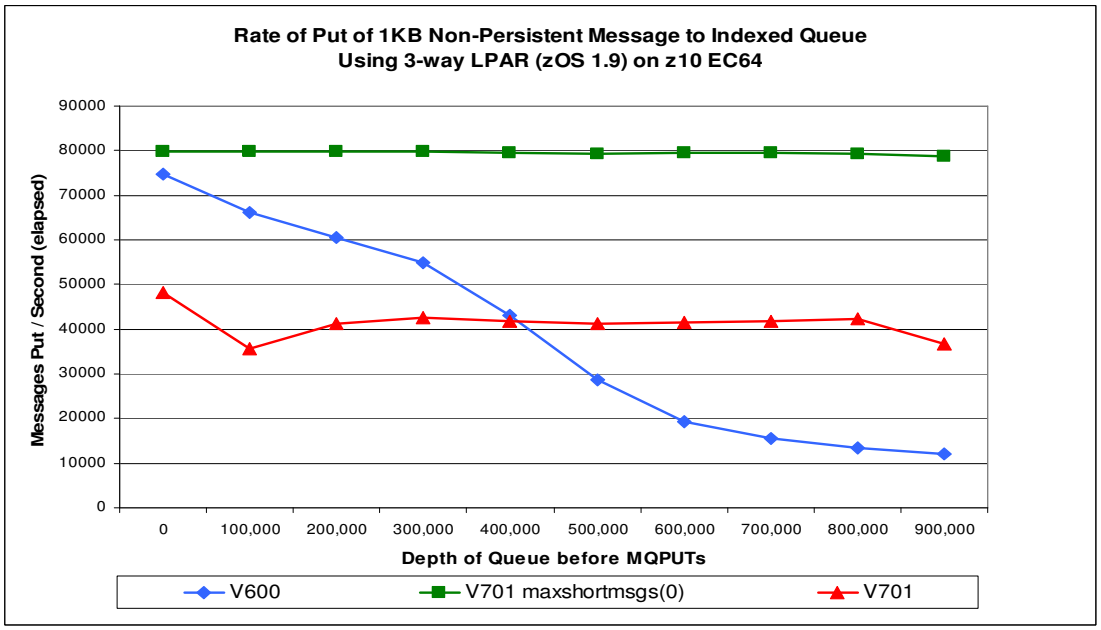
For version 6.0.0, note the increasing cost of the MQPUT as the depth of the queue increases.

For version 7.0.1, the cost of the MQPUT is flat, although the cost is less when the queue manager is running with "maxshortmsgs 0" – the difference being approximately 5 microseconds per message.

The costs shown are the total cost to the application putting the message plus the queue manager cost.



The following chart shows the rate at which the messages were put by a single batch application when the target indexed queue held an increasing number of messages.



When putting messages to the version 7.0.1 queue manager (red line with diamonds), the first 200,000 messages are held in bufferpools. After that point, there is an increase in the disk I/O as messages are written to pageset.

Both version 6.0.0 and version 7.0.1 with “maxshortmsgs 0” are able to put 400,000 messages to the bufferpools before messages are put to pageset.

Cost of getting messages from indexed queues

The previous section has shown that the cost of putting messages to indexed queues is now a flat cost in version 7.0.1, unlike in version 6.0.0.

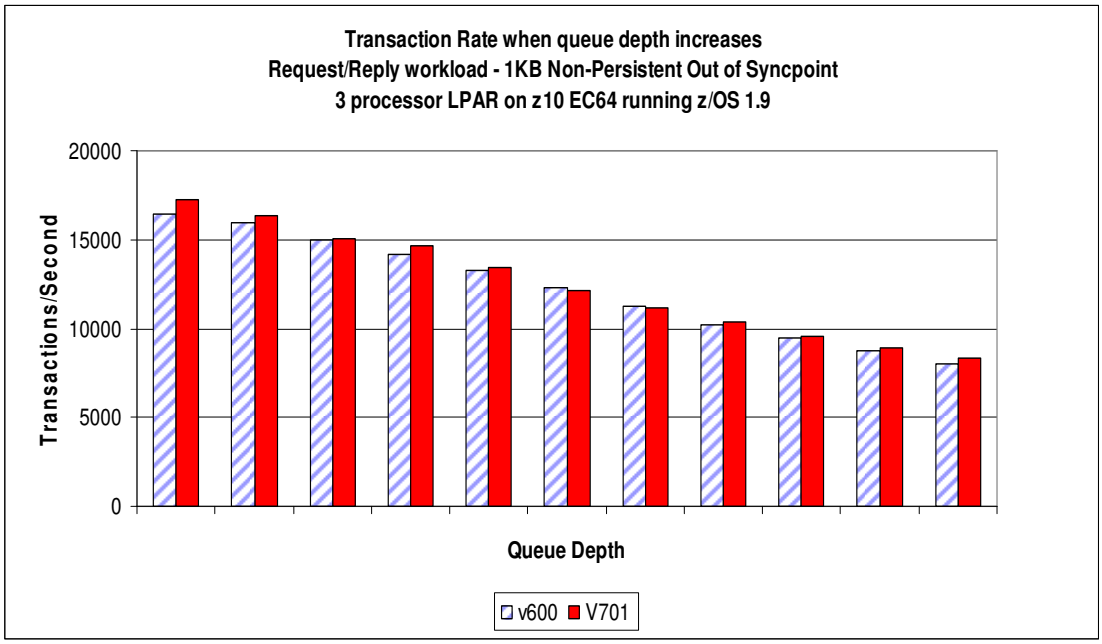
On an indexed queue, the deeper the queue, the more expensive the MQGET.

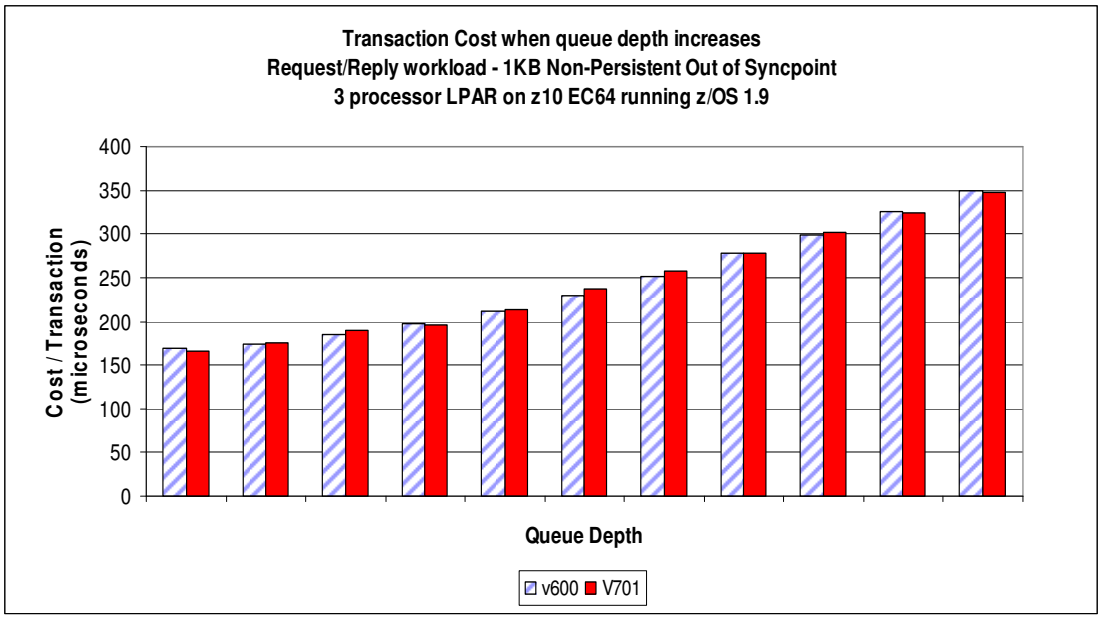
In a request/reply model with increasing depth of indexed queues, we see that the achieved transaction rate and cost per transaction for version 6.0.0 and version 7.0.1 is similar. The following 2 charts show this in more detail.

In the 2 sets of measurements, the queue managers are configured identically.

The queue managers are running with:
- TRACE(S) enabled
- TRACE(A) CLASS(3) enabled
- TRACE(G) disabled.

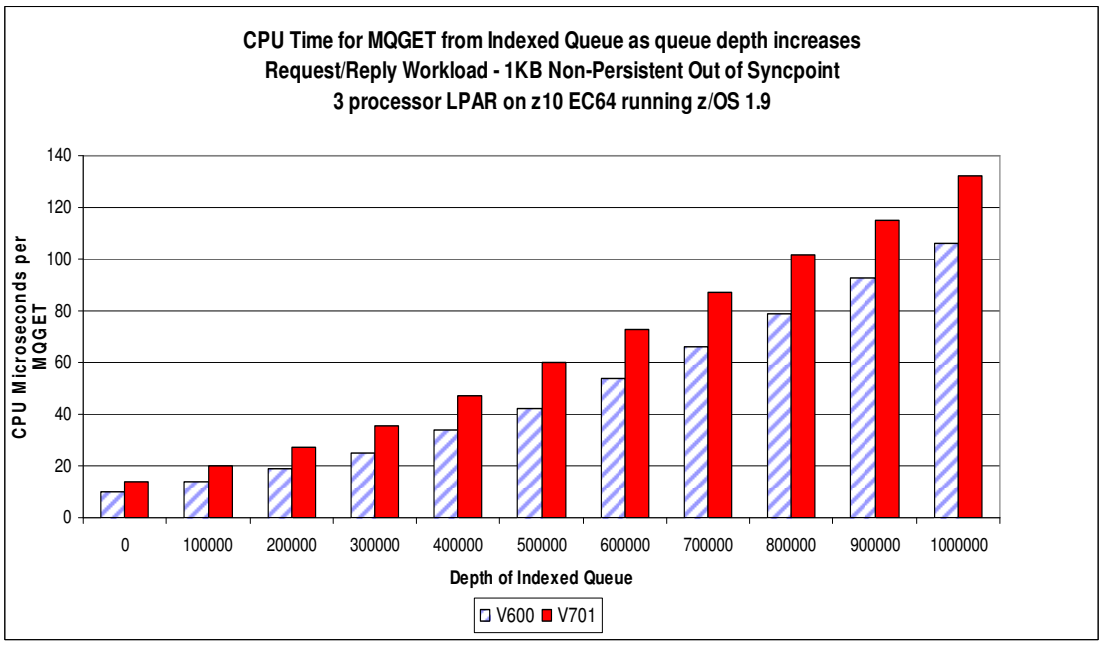
Prior to each iteration of the measurements, the indexed queue is initialised with increasing numbers of messages – from 0 to 1,000,000.





The previous 2 charts show that in a request/reply model using indexed queues, version 7.0.1 is keeping parity with version 6.0.0.

Since the cost of the MQPUT to the indexed queue is constant, it can be assumed that the cost of the MQGET is increasing. The following chart is taken from the SMF 116 data recorded during the measurements above.



The above chart indicates that the cost of the MQGET on version 7.0.1 does increase at a higher rate than in version 6.0.0.

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

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

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

```
CSQI007I @VKW7 CSQIRBLD BUILDING IN-STORAGE INDEX FOR <queueName>
CSQI006I @VKW7 CSQIRBLD COMPLETED IN-STORAGE INDEX FOR <queueName>
```

The queue manager is not available for work until all the indices are rebuilt.

- Unless CSQ6SYSP parameter QINDXBLD is set to NOWAIT
  - Any applications attempting to use the indexed queue where the index has not been rebuilt will wait until the index build has completed.

As indexed queues may be significantly deeper in version 7.0.1 than previously, the depth of the queues impacts the time taken to restart a queue manager.

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

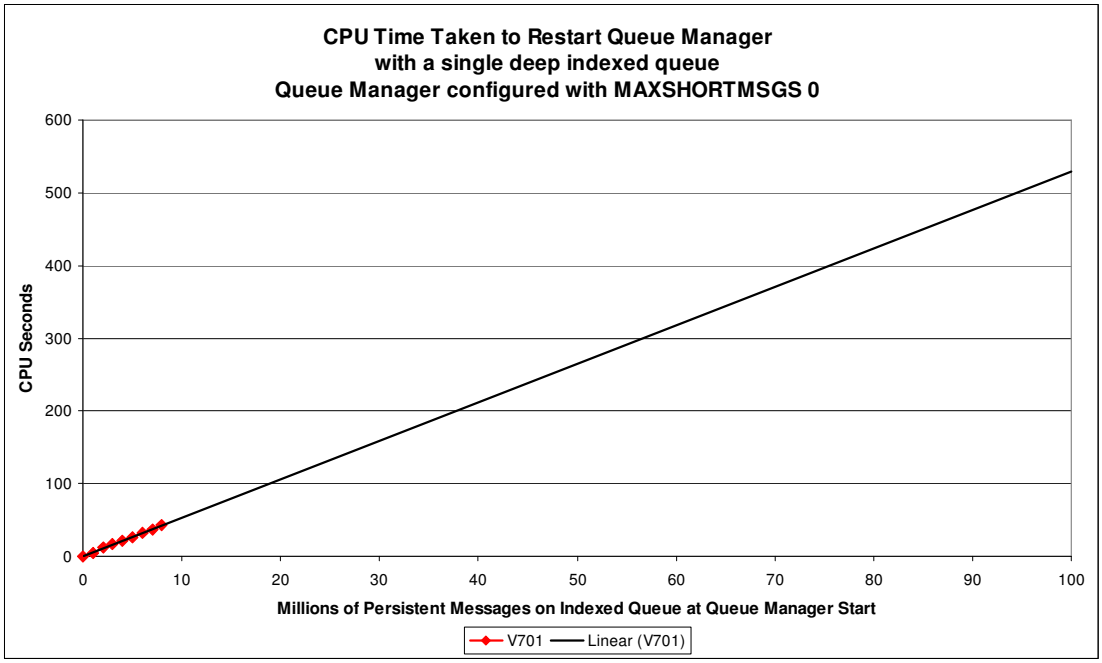
- If durable subscriptions exist, the SYSTEM.DURABLE.SUBSCRIBER.QUEUE may be rebuilt first.
- The deepest indexed queues are rebuilt next.

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

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

The Effect of a Single Deep Indexed queue upon Queue Manager Restart

The following chart shows the measured CPU cost to start a V7.0.1 queue manager when a single indexed queue has increasing depth. The queue manager has been configured with “MAXSHORTMSGS 0” to allow up to 8 messages of 100 bytes per 4K page.



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

In our measurements for the single queue, the index rebuilding process was able to use a single processor at approximately 15%.

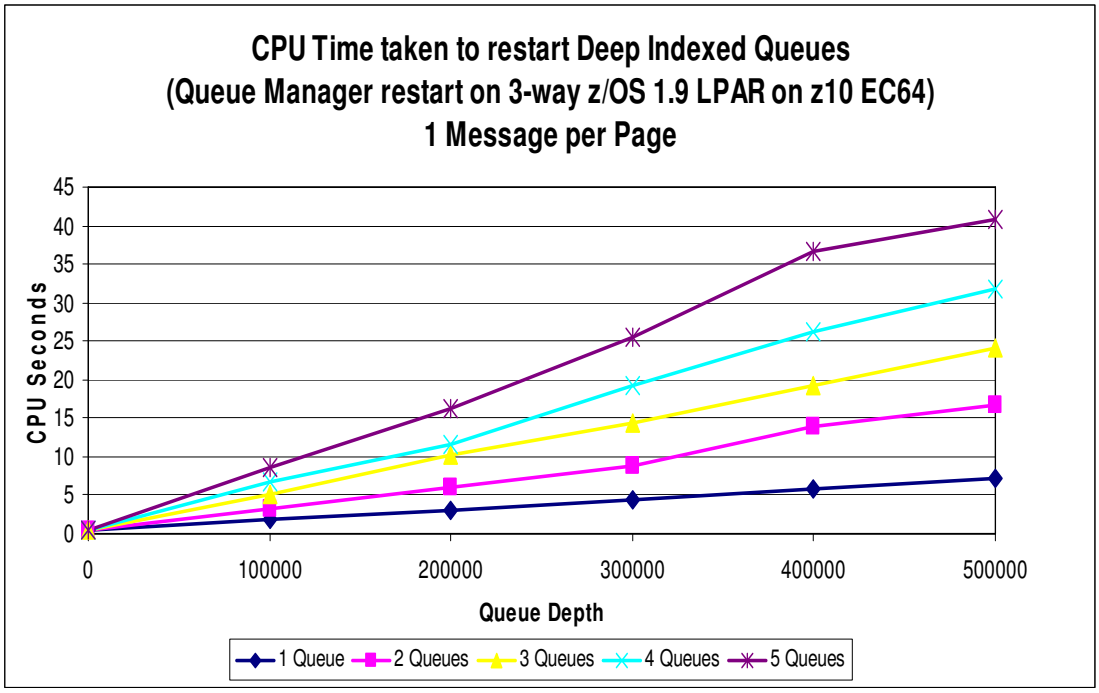
This means that for a queue with 8 million short messages on the queue, the CPU time taken was 42 seconds but the elapsed time was 280 seconds.

Using the trend line, we would expect that a queue with 100 million messages on would use 530 CPU seconds and would take approximately 1 hour to restart on our 3-processor LPAR of a z10 EC64.

The Effect of Multiple Deep Indexed Queues upon Queue Manager Restart

The following measurement shows the measured CPU cost to restart a queue manager with an increasing number of deep indexed queues – each on separate pagesets.

As the queue manager is not using "MAXSHORTMSGS 0", the 4GB pagesets are unable to hold more than 1 million messages of up to 3635 bytes.



The rebuilding of the indices is not a particularly CPU-intensive function so we are not constrained by having 3 processors available when re-building more than 3 queues concurrently. In this case the time taken to rebuild the queue indices is constrained by the rate at which the data can be read from DASD.

Security Manager

WebSphere MQ for z/OS version 7.0.1 uses 64-bit storage for security manager purposes. This releases storage within the queue manager for other purposes and means that effectively the number of user IDs that can access MQ resources is limited only by the amount of auxiliary storage available.

How much storage does a single user use?

When a user issues a sign-on to CICS followed by a single transaction involving MQ, there is an associated cost of approximately **8.80KB** of 64-bit storage for each user – which includes storage for a single queue. This is an approximate since the 64-bit storage is allocated in 1MB blocks.

Monitoring the real storage usage by using the SDSF© option "DA" (active users), we have found that the real storage usage increases for the queue manager address space increases by approximately 53% of the 64-bit storage used, i.e. 1.17 frames or **4.66KB** per user.

Whether using 1 or 50,000 user IDs, the queue managers 31-bit storage usage remains consistent – i.e. SQA, CSA, ESQA and ECSA usage are not impacted by security manager.

How much storage is required to access multiple MQ resources?

For each MQ resource accessed by a particular user, an average of 405 bytes of 64-bit storage is used.

Why is storage retained after security timeout has been reached?

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

The **security interval** is the time that passes between an MQ process checking the last use time of all the authenticated user IDs to determine whether the security timeout period has passed.

Use the "DISPLAY SECURITY" command to review the security timeout period i.e.

```
CSQH015I MQPB Security timeout = 54 minutes
CSQH016I MQPB Security interval = 12 minutes
```

In releases prior to version 7.0.1, once the first security timeout following the security interval had been reached, any memory used for an expired user was released and the queue managers address space real storage usage would decrease by the corresponding amount.

The security manager in version 7.0.1 uses a pooling principle. The storage is allocated when required and when it is no longer required, the storage is returned to the pool.

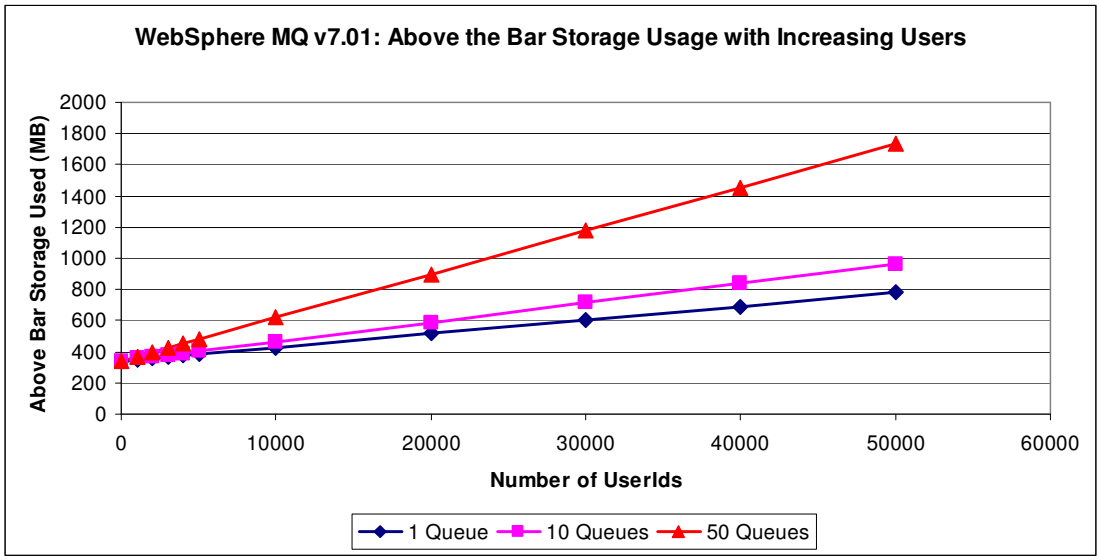
This re-using of the pooled memory uses less CPU cycles than the initial allocation. In our measurements we saw the re-use cost was approximately 10% less when 50,000 users performed activity against 10 MQ resources once all security data had been expired and the storage returned to the pool compared to the initial cost of the 50,000 users performing activity against 10 MQ resources.

Predicting Memory Usage with Increasing User IDs

The following chart shows the amount of 64-bit storage used within the queue managers address space for an increasing number of users.

There are 3 lines –

- Where each user uses 1 MQ resource
- Where each user uses 10 MQ resources
- Where each user uses 50 MQ resources.



The data was gathered using Workload Simulator "WSim" running CICS sign-on followed by 1, 10 or 50 serialised transactions to put and get a non-persistent message from up to 50 queues.

As previously reported, each user ID that uses an MQ resource that has security enabled will use 8.80KB of 64-bit storage. Therefore if a system has 100,000 users that each access 1 MQ queue, we can predict that security manager will use 859MB of 64-bit storage.

We can also predict that 53% of this 859MB will be used as real storage utilisation for the queue manager, unless the operation system needs to steal the storage for other tasks.

Log Compression

In version 7.0.1 of WebSphere MQ for z/OS, it is possible to enable log compression at a queue manager level. This means that for each queue manager on a system, log compression is either enabled or disabled.

Log Compression Measurements
We measured CPU costs and effect on throughput for 5 idealised models.
The first measurement was a baseline where log compression was not enabled.
All subsequent measurements ran with log compression enabled.

The models are described as below:

1. Log compression not enabled. Contents of message are irrelevant with regards to log compression.
2. Message data totally incompressible using RLE.
3. Message data compressible to approximately 90% of original length (i.e. message data is approximately 10% compressible).
4. Message data compressible to approximately 50% of original length.
5. Message data compressible to approximately 10% of original length.

The 5 models were measured using messages of the following sizes:

- 100 bytes, 1KB, 4KB, 100KB, 1MB.

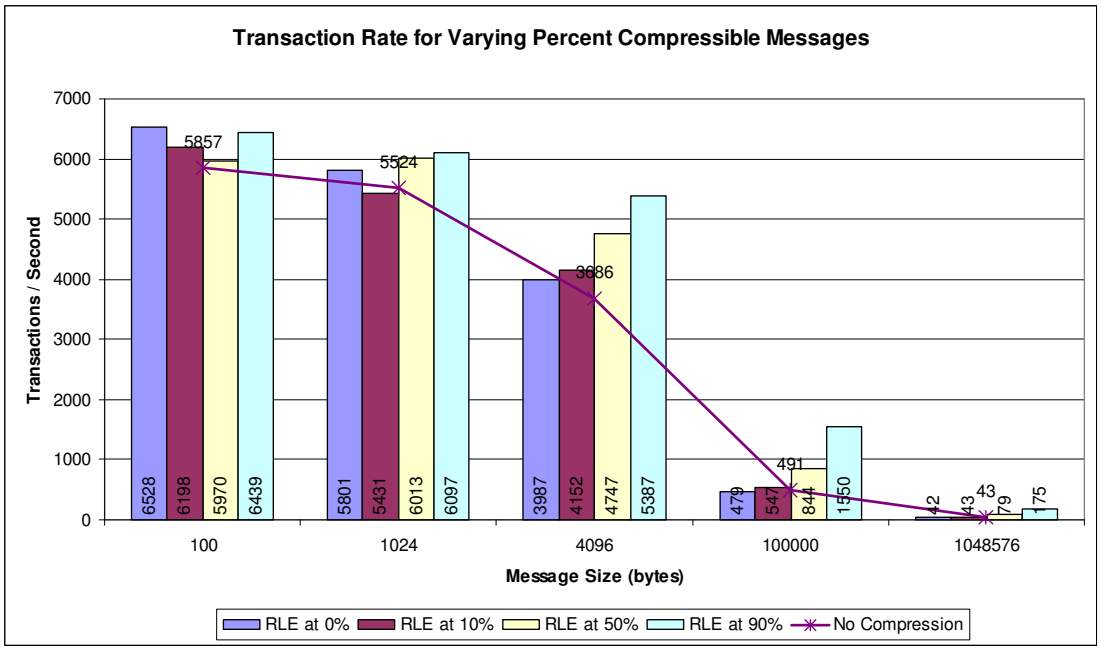
A request/reply-type scenario was run, where a transaction is defined as:

- Requester puts message and commits
- Server get and put message to indexed reply-to-queue and commit
- Requester gets message by index and commit.

A fixed number of applications (60 batch requesters, 10 batch servers) are run on a single z/OS 1.9 image with 3 dedicated processors on a 2097-EC64.

Measurements were run a single queue manager with:

- TRACE(A) CLASS(3) enabled
- TRACE(S) enabled
- TRACE(G) disabled
- SMF active.



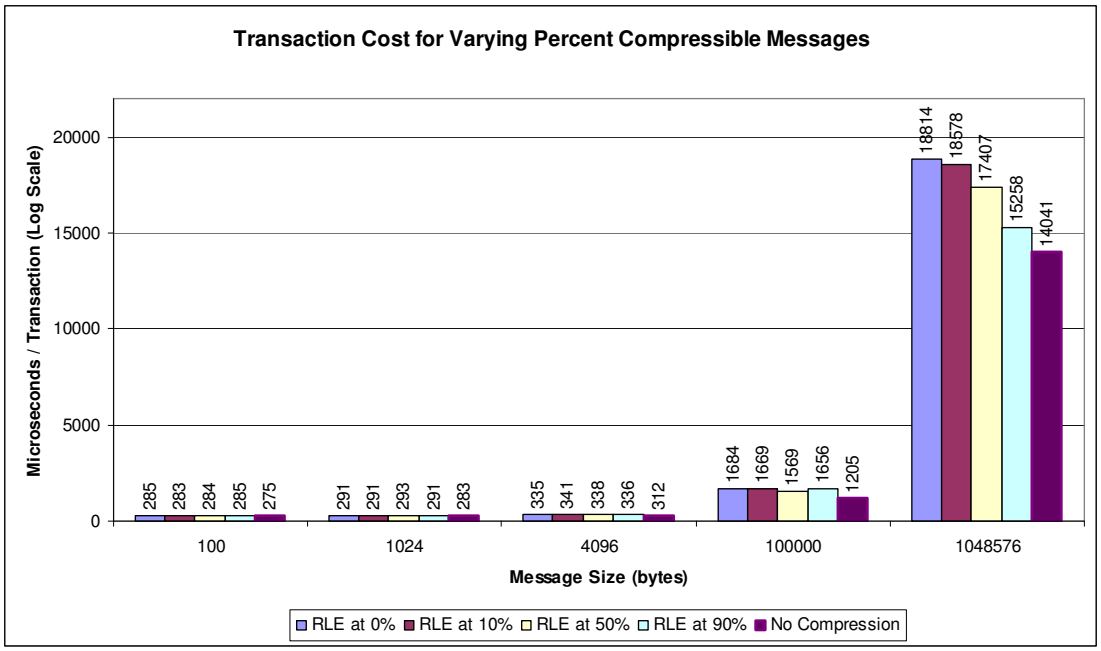
As can be seen in the above chart, using RLE compression has enabled the throughput to increase. For example, comparing the transaction rate for 100KB messages:

% Compressible	Transactions / Second	Delta % over "none attempted"
None Attempted	491	
0	479	-2.4
10	541	10.2
50	844	71.9
90	1550	315.6

In the above table, it can be seen that by compressing the 100KB message that contains 90% compressible data, we are able to increase the transaction rate by over 3 times.

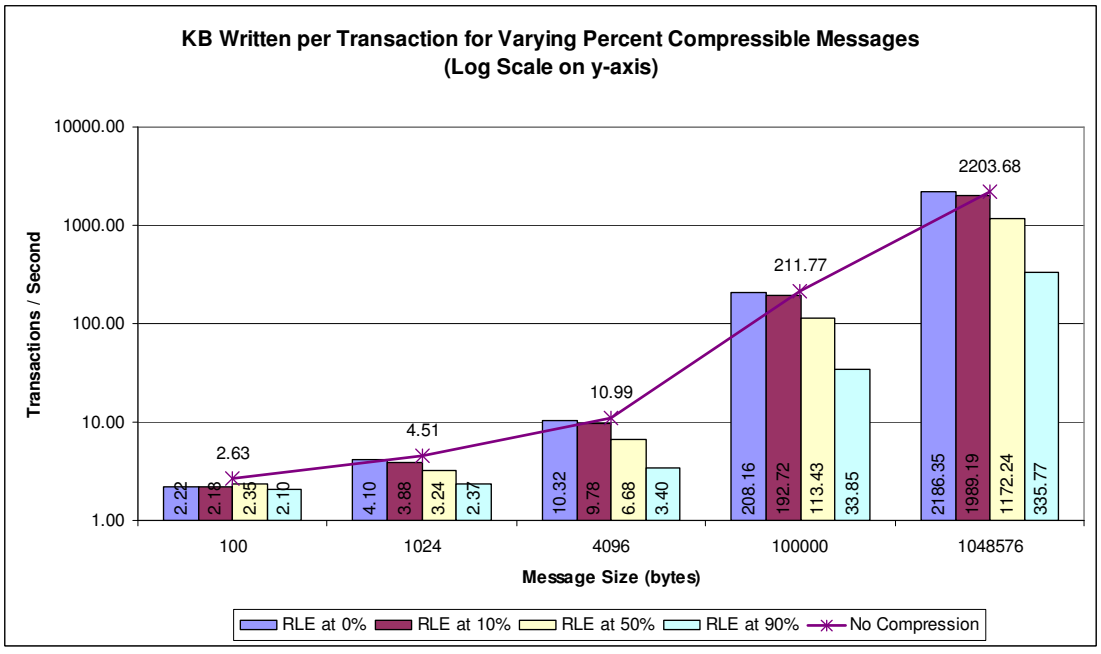
Attempting to compress messages regardless of whether or not the message is compressible does not come at zero cost.

In the following chart, we can see that the cost of attempting to compress messages is similar no matter how compressible the data is, but it is cheaper to process message containing 90% compressible data compared to a message with 50% compressible data.



The following chart plots the amount of data written to MQ logs based on the message size and the percentage compressible.

For small messages with little opportunity for data compression, log compression is still able to save 0.33 KB per transaction by compressing MQ header structures.



The previous chart shows the amount of data being written to the log datasets for each transaction.

In the measurements run, a transaction is:

- PUT+COMMIT (requester)
- GET+PUT+COMMIT (server)
- GET+COMMIT (requester)

Comparing the data for 1MB messages:

Compressible Message	Amount of log data written per transaction in MB
No	2.15
50%	1.14
90%	0.33

How do I know if Log Compression will help?

To help determine whether log compression is providing any benefit, the Accounting and Statistics records have been updated to provide information on the success of data compression.

The SMF 115 record includes the data and can be viewed by running CSQW1150.

The following diagram shows a sample data compression report generated by CSQW1150, taken following a workload involving 100KB messages that were approximately 50% compressible.

```
z/OS:MV25  MQ QMGR:VKW8  Time: 2009227 05:32:32.25
Log manager     : QJST
  Write_Wait          0,  Write_Nowait  22417134,  Write_Force       1729,  WTB          0
  Read_Stor          12,  Read_Active          0,  Read_Archive         0,  TVC          0
  BSDS_Reqs        3037,  CIs_Created    2731415,  BFWR             319241,  ALR          0
  ALW                 0,  CIs_Offload          0,  Checkpoints          0
  WUR                 0,  LAMA                 0,  LAMS                 0
  Write_Susp     319215,  Write_Reqs      107092,  CI_Writes       5472360
  Write_Serl          0,  Write_Thrsh          8,  Buff_Pagein          0

  Data compression  : 1
  Comp_Req      4905830,  Comp_fail            0,  Decomp_req           1, Fail         0
  Compression:   Before      19788652806,  After       10272387011  48%
  Decompression: Before              487,  After               878  44%
  Data compression  : 2
  Comp_Req            0,  Comp_fail            0,  Decomp_req           0, Fail         0
  Compression:   Before                0,  After                 0   0%
  Decompression: Before                0,  After                 0   0%
  Data compression  : 3
  Comp_Req            0,  Comp_fail            0,  Decomp_req           0, Fail         0
  Compression:   Before                0,  After                 0   0%
  Decompression: Before                0,  After                 0   0%
```

Only the first data compression section is used in version 7.0.1.

"Comp_fail" is where the message is larger after message compression has been attempted.

Compression Before / After – if the numbers are similar, log compression has achieved little benefit. In the above example, the amount of data has been compressed to 51.9% of its original size, i.e. a compression of 48%.

Select on Get

In WebSphere MQ for z/OS version 7.0.1 it is possible to specify a message selection criteria at the time the MQOPEN is issued.

Using this function ensures that all messages received by the application match the selected criteria and that filtering does not have to be performed at the application, whether the application is connected locally or as a client.

An optimisation has been added in version 7.0.1 to allow the application to use Correlation ID as the selector.

Using Correlation ID as a Selector

Selection using *correlationId* or *messageId* follows an optimised path through WebSphere MQ for z/OS version 7.0.1 and the selection occurs on the server-side (i.e. the queue manager). This gives better selection than when using arbitrary selectors.

To use the optimised path, the *correlationId* must be prefixed with "ID:" and must be formatted correctly as 24 bytes represented as a hexadecimal string (of 48 characters). Failure to adhere to this results in the selection using the more expensive client-side methods.

```
JMS Example:

Session.createConsumer(
  destination,
  "JMSCorrelationID='ID:574d51373053616d706c65436f7272656c6174696f6e4944'");
```

In the above example, the hexadecimal represents a 24-byte ASCII string "WMQ70SampleCorrelationID".

Use of the provider-specific "ID:" tag is applicable to only the *messageId* and the *correlationId* fields and is of practical use only with correlation identifiers.

The safest way to generate a correct identifier is to use *JMSMessageSet.setJMSCorrelationIDAsBytes*. This allows the formatted version to be set by *getJMSCorrelationID*. The number of bytes input should not be more than 24 otherwise the identifier will be truncated.

```
JMS Example:

Message.setJMSCorrelationIDAsBytes(
  "WMQ70SampleCorrelationID".getBytes("UTF8") );
Session.createConsumer(
  destination,
  "JMSCorrelationID='" + message.getJMSCorrelationID() + "'");
```

Changing the *correlationId* (or indeed any selector) that is being used to match against, requires the old MessageConsumer to be discarded and a new MessageConsumer to be opened. This is an expensive operation if it is performed for every message that is processed since it involves closing and re-opening the underlying queue.
For this reason, it is suggested that a unique *correlationId* for each client is used, rather than the more typical design pattern of using the *messageId* of a sent message as the *correlationId* of the reply message. Alternatively a temporary queue could be used for each client.

Local Bindings using Local Queue with Single Queue Manager

A message put to a queue may or may not have been put with message properties. Similarly a message gotten from the queue using a selection criteria on MQGET need not have properties, provided the selection criteria is the *correlationId*.

The act of adding the first message property to our message added 12 microseconds to the cost of the MQPUT. The next 14 properties added to the message added approximately 1 microsecond per property to the cost of the MQPUT.

This suggests that if the required message can be identified using the *correlationId*, then the cost of processing the message is significantly less.

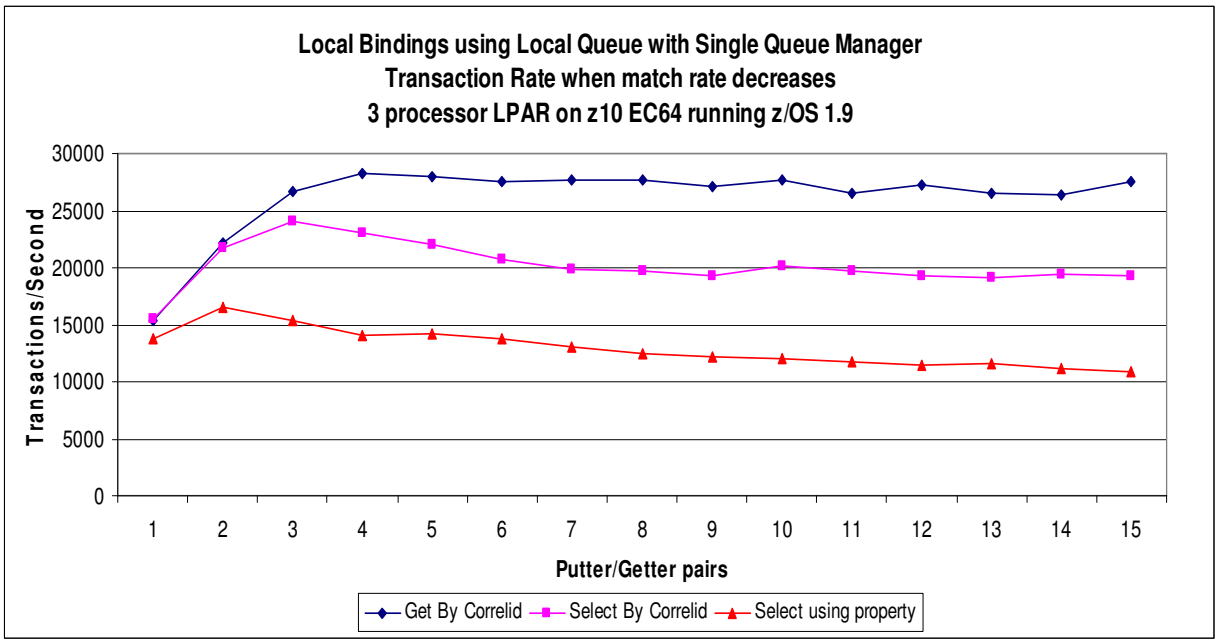
Measurements:

The following charts show the maximum sustainable throughput using 1KB non-persistent messages with batch application putting and getting the messages.

The test is configured thus:

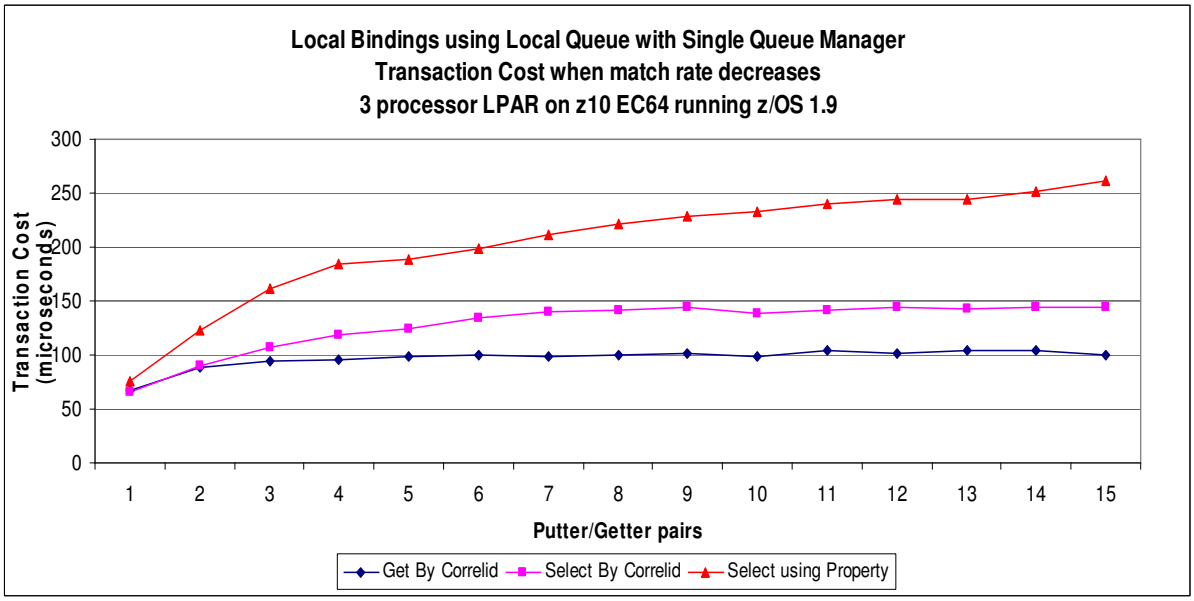
- A single pair of queues. Both are indexed with INDXTYPE(CORRELID). One is a request queue and the other is a reply queue.
- The getter applications are getting the messages as fast as possible. The message is gotten and a reply message is put out of syncpoint.
- The putter applications put a message to the request queue and waits for a corresponding message on the reply queue, before repeating.
- The applications are written in C and run in batch to eliminate as many of the overheads incurred in a more complex environment, such as CICS or Java.
- All messages are put with 15 properties.
- The test begins with one putter and one getter application. The number of putters and getters increases by one until there are fifteen putter and fifteen getter applications.
- When there is one putter, all messages will be valid for the getter. As more putters are added, the less likely it is that any message will be valid for a particular getter, i.e. when there are 10 putters and 10 getting applications, there is a 10% chance that any message is for a particular getter.
- There are 3 measurements
  1. When the getter applications specify the correlation ID in the MQMD.
  2. When the getter applications specify the correlation ID in the MQOD SelectionString

3. When the getter applications specify a property in the MQOD SelectionString.



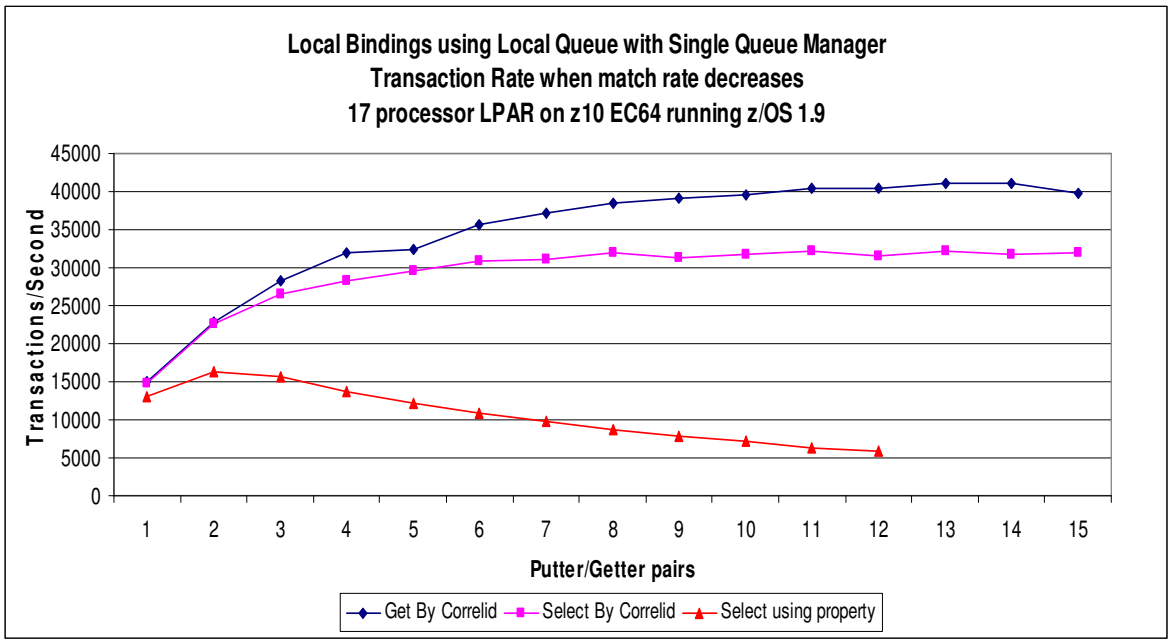
The above chart shows the achieved transaction rate.

Specifying the correlationID in the MQMD allows the best throughput rate. Using the SelectionString to specify the correlationID is significantly faster than selecting a message using a message property.



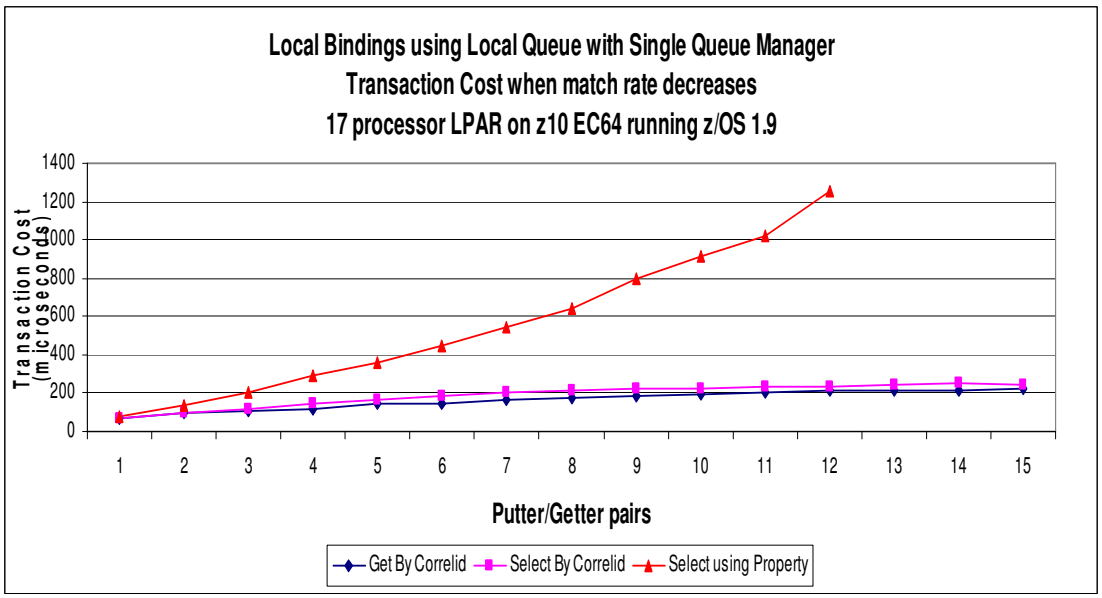
The above chart shows the cost per transaction in CPU microseconds for the described measurements. Essentially the transaction rates and transaction cost begin to flatten out as the machine runs close to capacity.

The following measurements are a repeat of the previous scenario, except the LPAR used has 17 processors available. This should remove any CPU constraints.



The above chart suggests that for "Get by Correlid", the maximum achievable transaction rate for a single queue has been reached.

The transaction rate for "Select using property" is decreasing as each message needs to be evaluated potentially by more and more getter tasks to determine if the particular message is required by the particular getter.



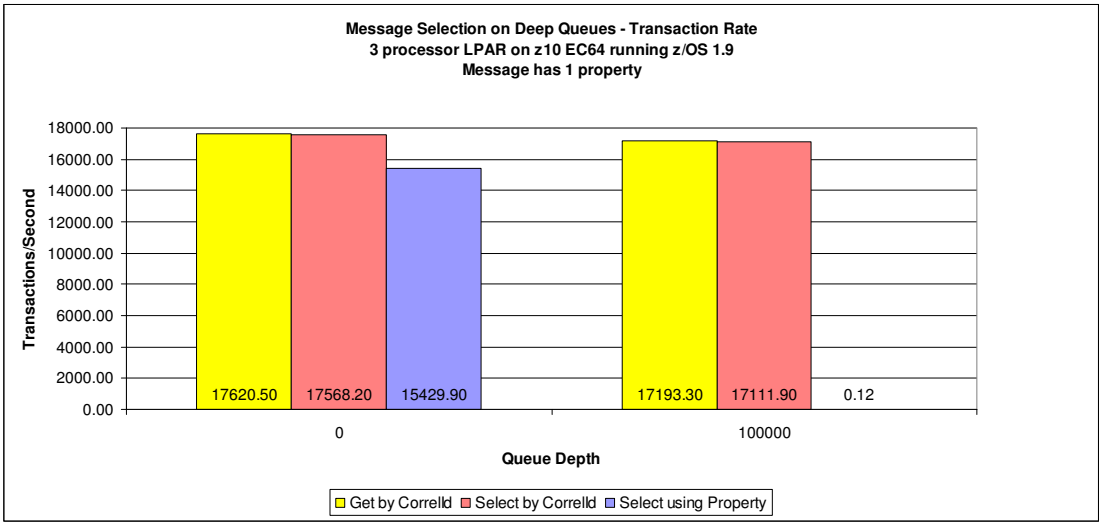
The above chart shows how the cost of additional getter tasks parsing more messages increases at approximately n-squared.

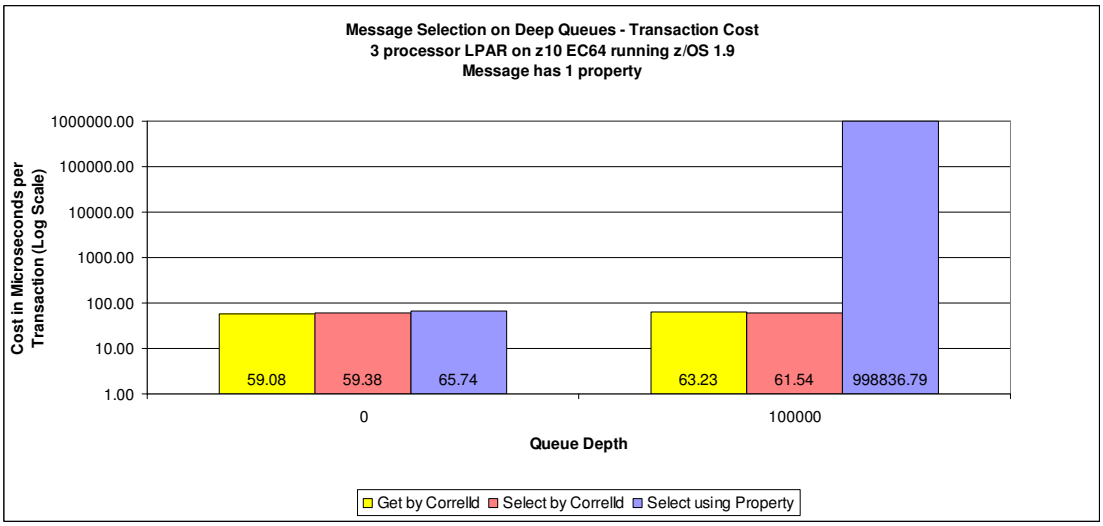
Does the depth of the queue affect message selection rates?

In the earlier section "Cost of getting messages from indexed queues" it has been shown that as the depth of the indexed queue increases, the cost of the MQGET increases and the transaction rate decreases.

When selecting messages using the correlationID either by the MQMD or the MQOD SelectionString, the transaction rate and cost are similar.

When selecting messages by specifying a selection string other than correlationID in the MQOD SelectionString, the selection process needs to re-scan **all** the messages on the queue, each time the MQGET is issued. The effect of this can be seen in the following 2 charts.





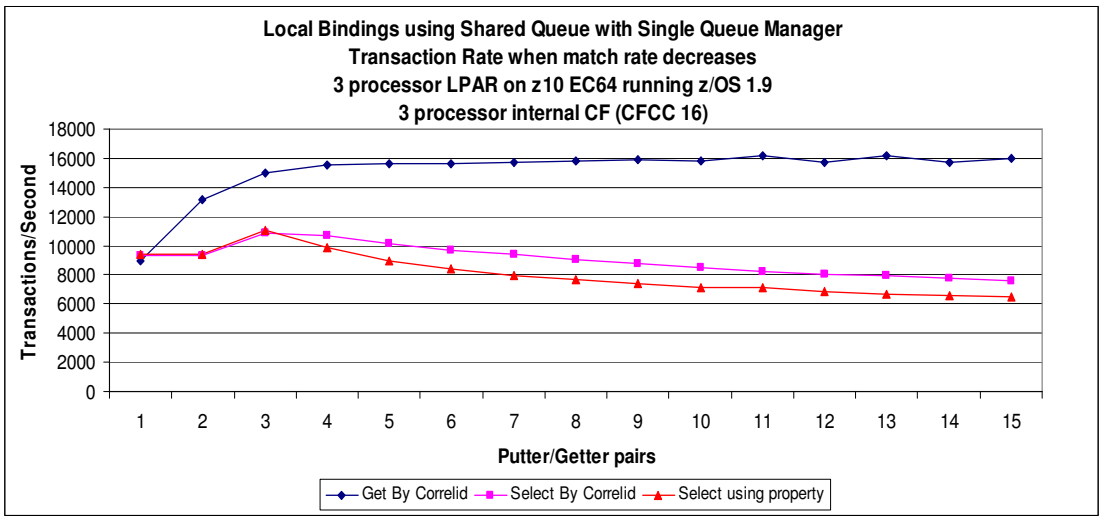
The above chart shows that when selecting using a property from a queue that has 100,000 messages that are not of interest to the selecting application, the cost of getting the required message is almost 1 CPU second, i.e. almost 1,000,000 CPU microseconds.

Local Bindings using Shared Queue with 1 Queue Manager in QSG

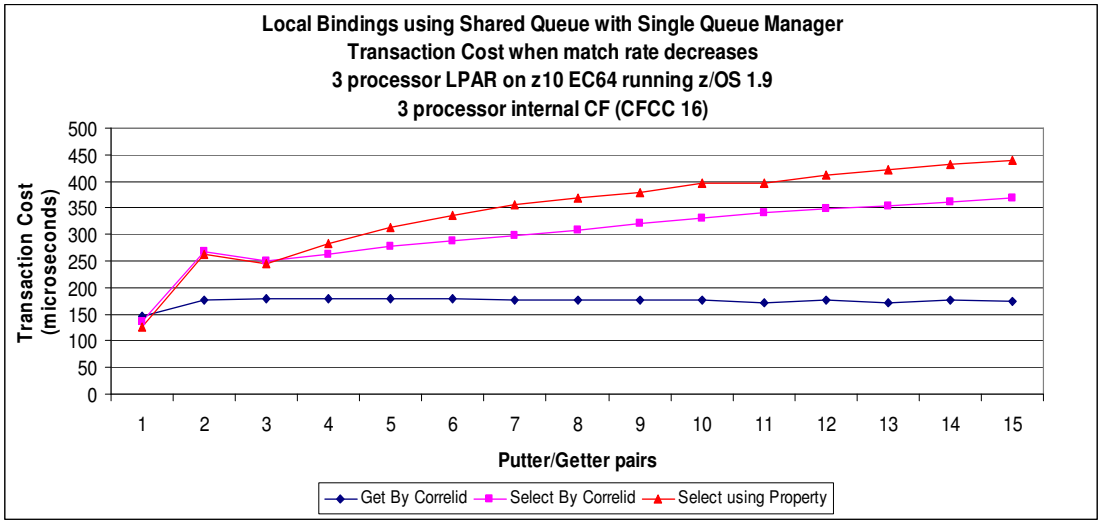
The following charts show the maximum sustainable throughput using 1KB non-persistent messages with batch application putting and getting the messages to and from shared queues.

The test is configured thus:

- A single pair of queues in a queue sharing group. Both queues are in the same coupling facility structure, also both are indexed using INDXTYPE(CORRELID). One is a request queue and the other is a reply queue.
- The getter applications are getting the messages as fast as possible. The message is gotten and a reply message is put out of syncpoint.
- The putter applications put a message to the request queue and waits for a corresponding message on the reply queue, before repeating.
- The applications are written in C and run in batch to eliminate as many of the overheads incurred in a more complex environment, such as CICS or Java.
- All messages are put with 15 properties.
- The test begins with one putter and one getter application. The number of putters and getters increases by one until there are fifteen putter and fifteen getter applications.
- When there is one putter, all messages will be valid for the getter. As more putters are added, the less likely it is that any message will be valid for a particular getter, i.e. when there are 10 putters and 10 getting applications, there is a 10% chance that any message is for a particular getter.
- There are 3 measurements
  1. When the getter applications specify the correlation ID in the MQMD.
  2. When the getter applications specify the correlation ID in the MQOD SelectionString
  3. When the getter applications specify a property in the MQOD SelectionString.



The above chart shows the achieved transaction rate. As with local queues, the MQGET using correlationID is significantly faster than using the select on get functionality.



The above chart shows the transaction cost for the 3 different measurements. The costs include both the requester and server applications as well as the associated queue manager costs for each transaction.

Local Bindings using Shared Queue with 2 Queue Managers in QSG

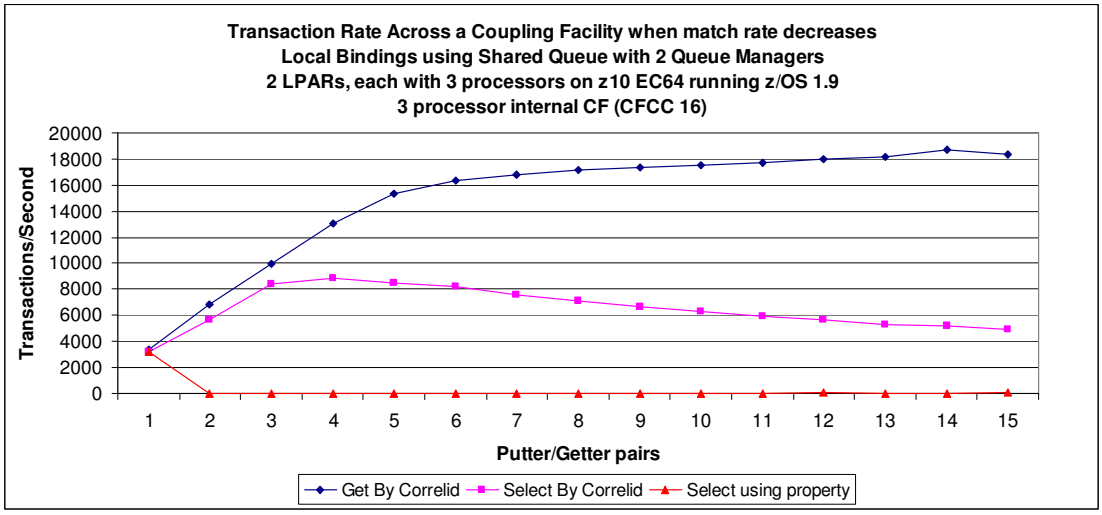
The following charts show the maximum sustainable throughput using 1KB non-persistent messages with batch application putting and getting the messages to and from shared queues.

In this scenario, there are 2 queue managers in the queue sharing group on separate LPARs of a z10 EC64. Each LPAR has 3 dedicated processors and are running z/OS 1.9.

The test is configured thus:

- A single pair of queues in a queue sharing group. Both queues are in the same coupling facility structure, also both are indexed using INDXTYPE(CORRELID). One is a request queue and the other is a reply queue.
- The getter applications are getting the messages as fast as possible. The message is gotten and a reply message is put out of syncpoint. The getter application is on LPAR 2.
- The putter applications put a message to the request queue and waits for a corresponding message on the reply queue, before repeating. The putting application is on LPAR 1.
- The applications are written in C and run in batch to eliminate as many of the overheads incurred in a more complex environment, such as CICS or Java.
- All messages are put with 15 properties.
- The test begins with one putter and one getter application. The number of putters and getters increases by one until there are fifteen putter and fifteen getter applications.

- When there is one putter, all messages will be valid for the getter. As more putters are added, the less likely it is that any message will be valid for a particular getter, i.e. when there are 10 putters and 10 getting applications, there is a 10% chance that any message is for a particular getter.
- There are 3 measurements
  1. When the getter applications specify the correlation ID in the MQMD.
  2. When the getter applications specify the correlation ID in the MQOD SelectionString
  3. When the getter applications specify a property in the MQOD SelectionString.



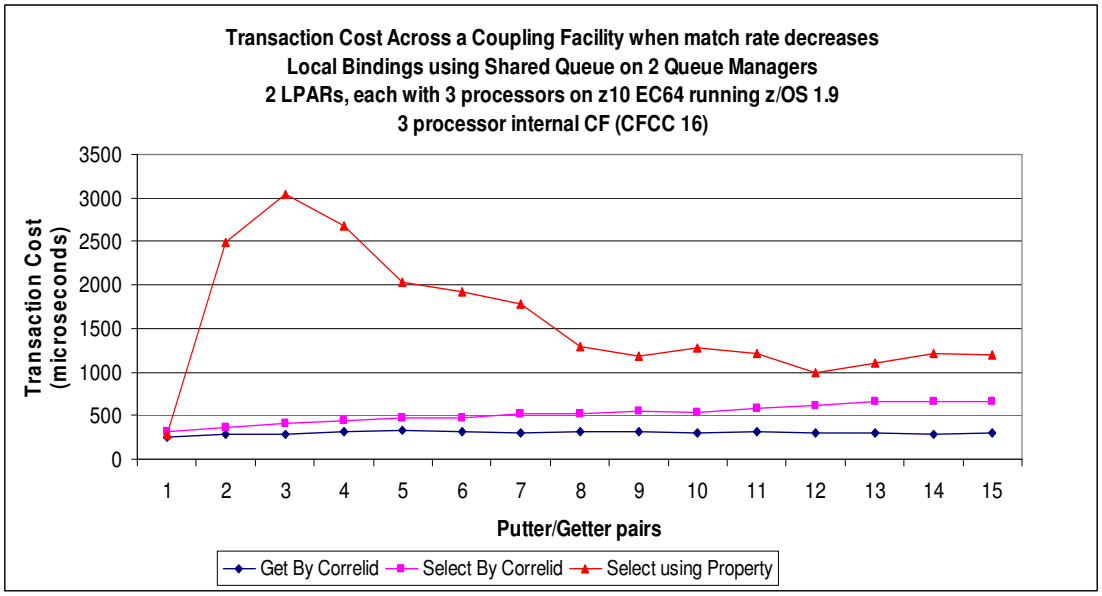
The above chart shows the achieved transaction rate when using message selection across a coupling facility.

The “Get by Correlid” measurement is optimised such that the Coupling Facility is able to pass the appropriate message to the getting application.

The “Select using Correlid” measurement sees the getting queue manager given the message with the appropriate correlationID but then parses the message to ensure that the JMS correlationID has not been overridden.

In the “Select using Property” measurement, the getting queue manager has to get every message from the queue and parse the message to determine whether the message is required by the particular getter task.

The following chart shows the cost per transaction. Whilst the cost of the "Select by Correlid" measurement tracks the "Get by Correlid", the "Select using Property" costs are significantly higher. This high cost has been explained above.

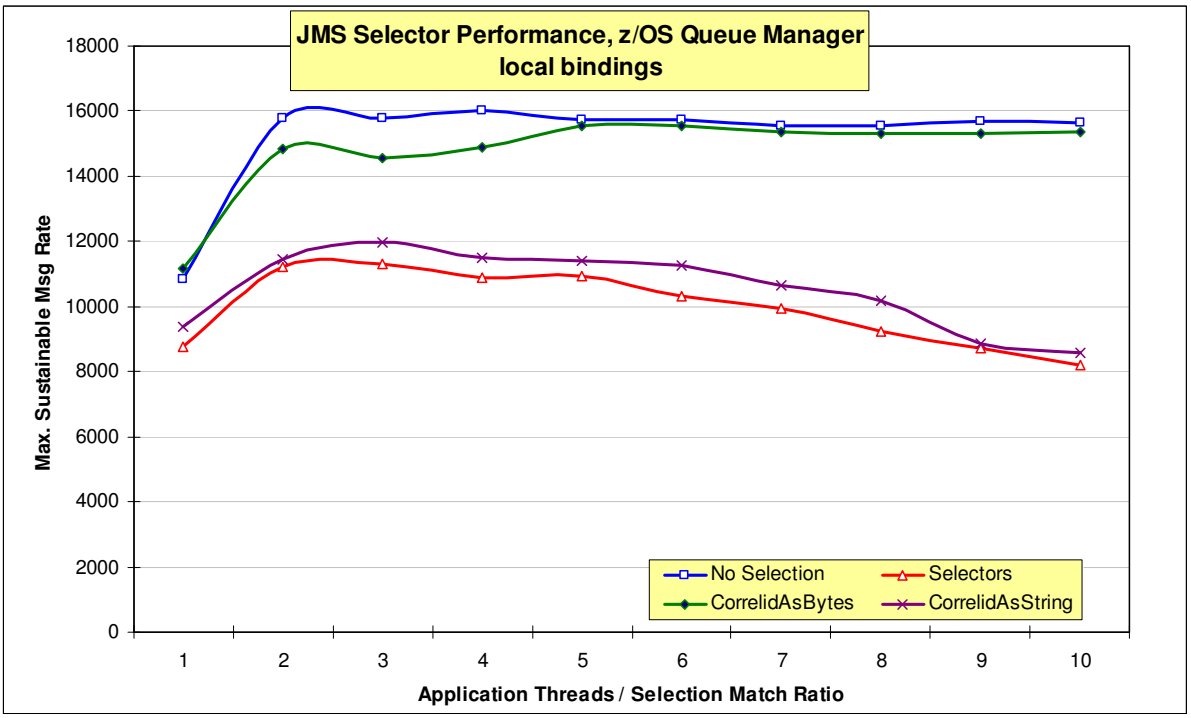


Local Bindings with JMS Selectors and Correlation ID

The following chart shows the maximum sustainable throughput using 1KB non-persistent messages with JMS applications putting and getting the messages.

The test is configured thus:

- A single local queue is used.
- The JMS applications are running in the USS environment.
- The getter application is getting messages as fast as possible.
- The putter application is throttled to ensure the queue depth does not increase to a point where the messages are out of buffer pools.
- The test starts with 1 putting application and 1 getting application. The number of application increases by 1 until there are 10 putting and 10 getting applications.
- When there is 1 putting application, all messages put will be valid for the getter. As more putters are added, the less likely it is that any message will be for any particular getter, i.e. when there are 10 putters and 10 getters, there is only a 10% chance that any message is for a particular getter.

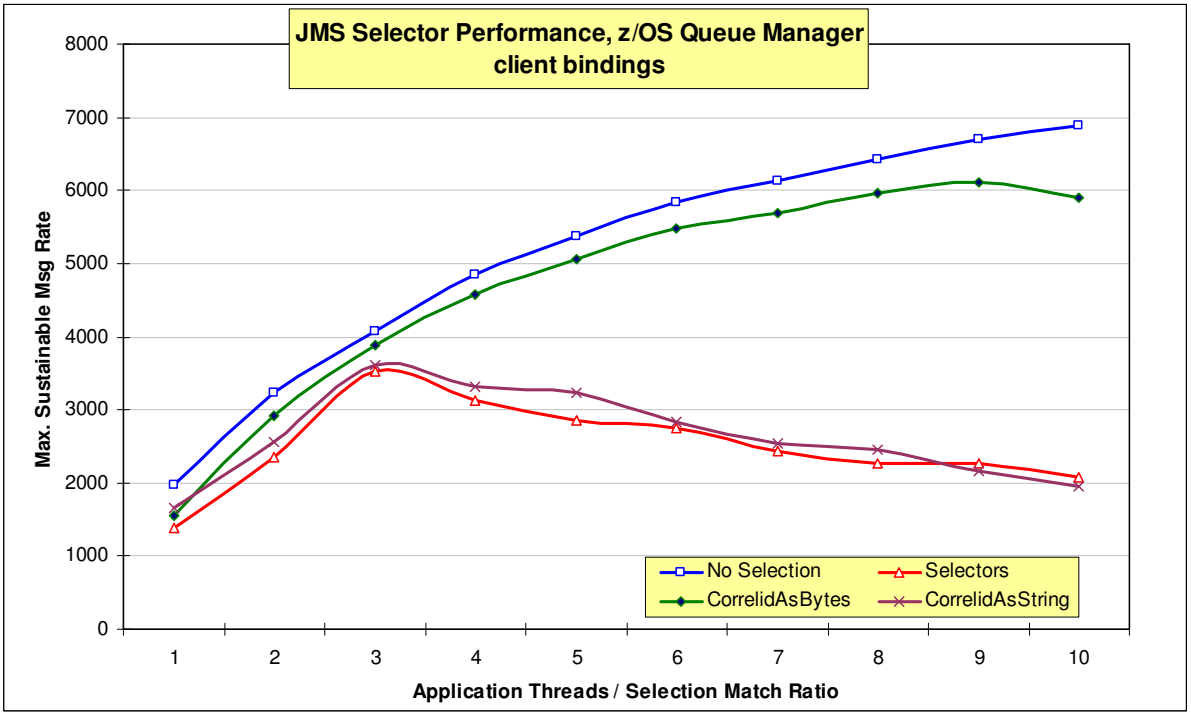


Client Bindings with JMS Selectors and Correlation ID

The following chart shows the maximum sustainable throughput using 1KB non-persistent messages with JMS applications putting and getting the messages.

The test is configured thus:

- A single local queue is used.
- The JMS applications are running on a client machine
- The getter application is getting messages as fast as possible.
- The putter application is throttled to ensure the queue depth does not increase to a point where the messages are out of buffer pools.
- The test starts with 1 putting application and 1 getting application. The number of application increases by 1 until there are 10 putting and 10 getting applications.
- When there is 1 putting application, all messages put will be valid for the getter. As more putters are added, the less likely it is that any message will be for any particular getter, i.e. when there are 10 putters and 10 getters, there is only a 10% chance that any message is for a particular getter.



Measurement Environment and Methodology

Hardware and Software

The hardware configuration was:

- **CPU:** 3-CPU logical partition (LPAR) of a z10 EC64 (2097-EC64). An internal CF with 3 floating processors was used for shared queue measurements.
- **DASD:** FICON-connected Enterprise Storage Server (ESS) Model F20.

Software levels were:

- z/OS 1.9
- WebSphere MQ v6 GA
- WebSphere MQ v7.0.1 pre-GA levels
- CICS CTS 3.2
- DB2 v9
- IMS v10
- Java 1.5

Client testing was performed on:

- 64-bit RHEL Linux on 4-way XEON 3.66 Ghz processor.

CPU cost calculations on other zSeries systems

CPU costs can be translated from a measured system to the target system on a different z/Series machine by using Large Systems Performance Reference (LSPR) tables. These are available at: http://www.ibm.com/servers/eserver/zseries/lspr/zSerieszOS.html

This example shows how to estimate the CPU cost for a zSeries 2064-1C5 where the measurement results are for a 2084-304:

1. The LSPR gives the **2064-1C5** an Internal Throughput Ratio (ITR) of 2.45
(this is for a "Mixed Workload", which we found best fits WMQ in our environment).

2. As the 1C5 is a 5-way processor, the single engine ITR is
2.45 / 5 = **0.49**

3. The "Mixed Workload" ITR of the **2084-304** used for the measurement is **3.60**.
The 304 is a 4-way processor. Its single engine ITR is
3.60 / 4 = **0.90**

4. The **2064-1C5** / **2084-304 single engine ratio** is
0.49 / 0.90 = 0.54 approx

this means that a single engine of a 2084-304 is nearly twice as powerful as that of a 2064-1C5.

5. Take a CPU cost of interest from this report, say **x** CPU microseconds (2084-304) per message, then the equivalent on a 2064-1C5 will be
x / 0.54 CPU microseconds/message

6. To calculate CPU busy, calculate using the number of processors multiplied either by 1000 (milliseconds) or 1000000 (microseconds) to find the available CPU time per elapsed second.

I.E. a 2064-1C5 has 5 processors so has 5,000 milliseconds CPU time available for every elapsed second.

So, for a CPU cost of interest from the report of 640 milliseconds on a 2064-1C5, the CPU busy would be:

640 / (5*1000) * 100 (to calculate as a percentage) = 12.8 %