Websphere MQ for zOS V5.3 and V5.3.1 Performance Report

Version 2.0

March, 2003

Dave Fisher

Keith Wilcock

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

Take Note!

Before using this report be sure to read the general information under "Notices" on page iv.

Second Edition, March 2003

This edition applies to WebSphere MQ for z/OS - Version 5 Release 3 and Version 5 Release 3.1.

Contents

Notices

This report is intended to help the customer understand the performance characteristics for Websphere MQ for zOS Version 5 Release 3 and Version 5 Release 3.1. The information is not intended as the specification of any programming interfaces that are provided by MQSeries. Full descriptions of the MQSeries facilities reported are available in the product publications.

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 "asis". 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 of the IBM Corporation in the United States or other countries or both:

- IBM
- z/OS
- MQSeries
- WebSphere MQ
- WebSphere
- DB2

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

Summary of Amendments

Date	Changes
June 2002	Initial release
March 2003	A rewrite of the expiry processing section
	A new section on V5.3.1 new feature - channel receive timeout
	Additions and changes are marked with revision bars

Preface

This report is intended to help the customer understand the performance characteristics for new features in Websphere MQ for zOS V5.3 and V5.3.1. It also provides some comparisons with the previous release for existing function. No updates to this document are planned. Ongoing, more general performance and capacity planning information is provided in SupportPac MP16 - *Capacity planning and tuning for MQSeries for z/OS*, which is periodically updated. This is available at http://www.ibm.com/webspheremq/supportpacs/

1.0 Performance highlights -WebSphere MQ for z/OS V5.3 & V5.3.1

Version 5 Relese 3.1 has no significant performance differences versus V5.3 for the same function. The most significant changes in V5.3.1 are for JMS within a WebSphere Application Server (WAS) for z/OS environment. WAS performance information is not supplied by WebSphere MQ.

This document reports on the performance of the following

- Persistent messages on shared queues. We have demonstrated throughput of more than 11000 persistent 1KB messages per second. Higher rates should be possible using enough queue managers each with enough log I/O rate capacity.

  Shared queues provide increased availability, scalability and pull workload balancing possibilities. Messages in a shared queue are stored in a Coupling Facility (CF). They can be accessed by multiple queue managers on multiple z/OS images in a sysplex which are part of the same Queue Sharing Group (QSG).
  See Shared queue persistent message performance on page 0.

  Shared queue persistent message backup and recovery performance is also addressed, see How often should CF structures be backed up? on page 0.

- The incorporation of changes otherwise available only as PTFs to V5.2.

  - Up to 30% improved throughput for persistent messages when using dual logging, see page 0.

  - Avoiding significant swapping of batch applications without making them non swappable, see page 0

- Most existing function performance is not significantly different to the previous release when run under the equivalent conditions. However there are various changes which positively affect performance in particular circumstances including

  - Maximum single queue throughput improved up to 25%, MQPUTs no longer outpacing MQGETs, see Maximum throughput through a single queue improved on page 0

  - Improved throughput and CPU cost when a buffer pool is too small, see Buffer pool usage improvements on page 0.

  - Faster Queue manager restart by reducing log reads see page 0.

  - Private indexed queue rebuild at restartsee page 0.

  - Log data sets can now be preformatted see page 0.

- New function performance

  - Secure Sockets Layer (SSL) channels, see page 0. Not a major cost, but you should review whether you need a Cryptographic Coprocessor Feature if you are using Triple Des encryption.

  - Message grouping, see page 0

  - JMS is available as a separately installable feature. A very brief overview of batch performance is given, this shows significant improvement versus the MA88 SupportPac made available in mid 2001. See JMS performance on page 0

  - Queue manager initiated expiry processing on page 0.

  - *<Receivetimeout>Receive Timeout on page 0.*

2.0 Shared queue performance overview

Shared queues have the following prerequisites

- OS/390 2.9 or later
- Coupling Facility level 9 or later
- DB2 6.1 or later

and constraints

- Maximum message size 63 KB (64512 bytes including all headers other than MQMD)
- Maximum of 8 million messages per CF. This 8 million limit is removed by 64-bit addressing which is available in coupling facility micro code level 12 running on 64-bit hardware.
- Maximum of 512 shared queues per CF list structure
- Maximum of 63 CF structures per Queue Sharing Group (QSG) within a maximum of 512 CF list structures per sysplex
- Maximum of 32 queue managers per QSG

Shared queue messages, whether persistent or non persistent, are not lost because of any (or all) queue manager failure or normal shut down. They are lost if the CF list structure containing them fails. Persistent messages can be recovered after such a failure from a fuzzy backup copy and the logs of all the queue managers in the QSG. WebSphere MQ has BACKUP and RECOVER commands to enable this.

2.1 CF link type and shared queue performance

Shared queue performance is significantly influenced by the speed of Coupling Facility (CF) links employed. There are, at the time of writing, 3 different types of CF links . These are known to the operating system (see the result of a ‘**D CF’** operator command) as

- CFP (CFS on 9672 range processors) - also known as HiperLink, can be upto 27Km.
- CBP (CBS on 9672 range processors) - also known as ClusterBus, can be upto 10 metres.
- ICP (ICS on 9672 range processors) - these are the highest speed links but are only available within the same box.

All link types can be present. The operating system generally selects the fastest currently available.

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

Some uses of the CF are always asynchronous in which case CPU cost is not dependent on link speed. Most WebSphere MQ CF accesses for messages <= 3626 bytes (excluding the MQMD) are potentially synchronous.

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

It is a general parallel sysplex recommendation that CF utilization does not normally exceed 50%. This is to allow for the possibility of one CF temporarily taking over the work of another. RMF reports 'AVERAGE CF UTILIZATION (% BUSY)' .

If there is contention on a link then synchronous CF accesses can be changed by the operating system into asynchronous. Furthermore, z/OS 1.2 has introduced heuristics which can change any potentially synchronous call to asynchronous based its own observed internal performance variables.

The above factors mean that performance of shared queue is very sensitive to the overall workload and the particular CF configuration.

For example, as previously reported in the V5.2 Performance Report , our measurements of thin client driven request/reply using shared queues with 1000 byte non persistent messages are an example of the effect of the different link types. The overall CPU cost compared to V2.1 using non-shared queues increased by

- about 13% when using ICS links,
- about 16% when using CBS links
- about 34% when using CFS links.

Such CPU cost differences are highly workload dependent but tend to be greatest for short messages. These results, obtained on a 9672 system indicated that CBS links give performance more towards the faster and cheaper ICS end of the scale. V5.3 measurements in this document used either CFP links or CFP links plus an ICP link. No measurements using CBP links have been done.

2.2 Shared queue setup considerations

2.2.1 How many CF structures should be defined ?

You need at least two CF structures. One is the CSQ_ADMIN structure used by WebSphere MQ itself, any others are application structures used for storing shared queue messages.

For best performance we recommend using as few CF application structures as possible.

- There is a maximum of 512 shared queues per CF list structure. We have seen no significant performance effect when using a large number of queues in a single application structure.
- There is a maximum of 63 CF structures per Queue Sharing Group (QSG) within a maximum of 512 CF list structures per sysplex

Using two CF application structures for the locally driven request/reply workload increased the CPU cost by 21% (See the V5.2 Performance Report) compared to using one CF application structure.

Only if all the MQGETS and MQPUTS are out of syncpoint are the costs of using one or multiple application structures the same. Since most business applications are going to involve some unit of work processing use only a single CF application structure if possible.

2.2.2 What size CF structures should be defined ?

2.2.2.1 CSQ_ADMIN

This CF structure does not contain messages. It should usually be left at the default (and minimum) of 10000KB.

The WebSphere MQ command DIS CFSTATUS(CSQ_ADMIN) shows the maximum number of entries, for instance ENTSMAX(9362), and the current number of entries used, for instance ENTSUSED(53). We recommend you should allow 1000 entries for each queue manager in the queue sharing group. So our example is adequate for 9 queue managers in a QSG using a CF at CFCC level 10. Each successive CF level tends to need slightly more control storage for the CF's own purposes, so ENTSMAX is likely to decrease each time your CF level is upgraded.

2.2.2.2 Application structures

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

- Estimate message size (including all headers)
- Round up to 256 byte boundary (subject to a minimum of 1536 bytes in OS/390 V2.9)
- Multiply by maximum number of messages
- Add 25% (for other implementation considerations, this percentage can be much greater for application structures smaller than 16MB)
- Use this result for INITSIZE in the operating system CFRM (Coupling Facility Resource Manager) policy definition
- Consider using a larger value for SIZE in the CFRM policy definition to allow for future expansion. See *Increasing the maximum number of messages within a structure* on page 0.

Each successive CF level tends to need slightly more control storage for the CF's own purposes, so we recommend a minimum CF application structure size of 16MB.

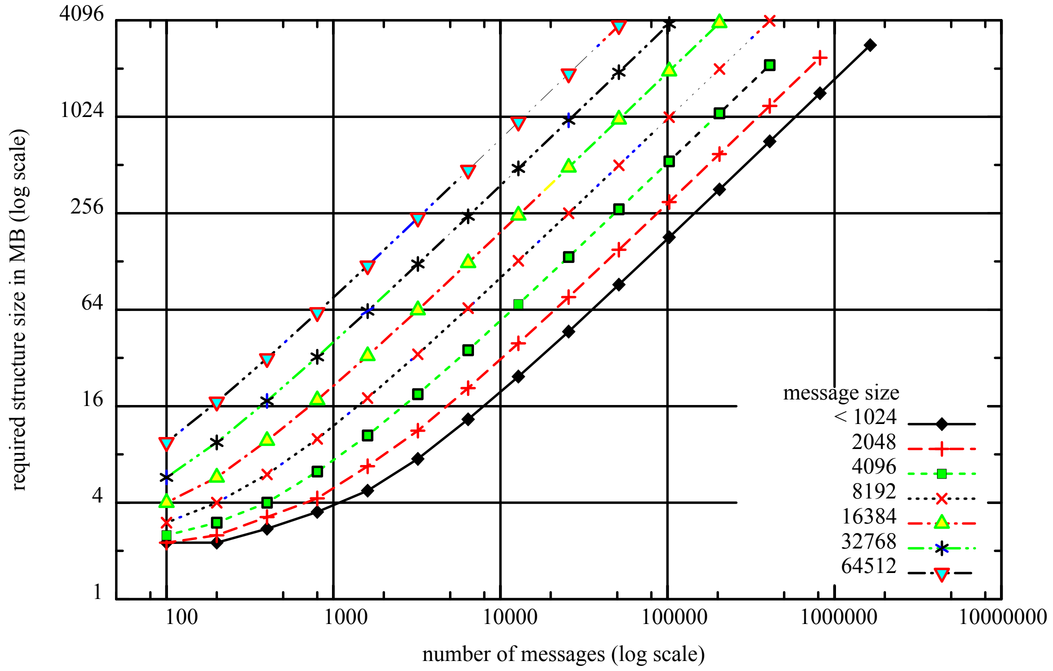
The following CFRM policy definition of an approximately 0.5GB CF list structure is typical of those used for our measurements.

```
  STRUCTURE NAME(PRF2APPLICATION1)     /* PRF2 is the QSG name */
          SIZE(1000000)
          INITSIZE(500000)
          PREFLIST(S0CF01)
```

Note that when running on OS/390 V2.9 the structure will not expand automatically beyond the INITSIZE even if SIZE has a larger value. See WebSphere(R) MQ for z/OS(TM) System Setup Guide Version 5 Release 3 Document Number SC34-6052 for details of MQ definitions.

To get some idea of how many messages you can get for a particular CF application structure size consult the following chart where 'message size' includes the user data and all the headers except the MQMD. For example, you can get about 750 messages of 64512 bytes(63KB) in a 64MB structure or nearly 50000 16KB messages in a 1GB structure. For all message sizes from 0 to 1212 (1536 - length(MQMD)) you can get about 275000 messages in a 0.5GB structure.

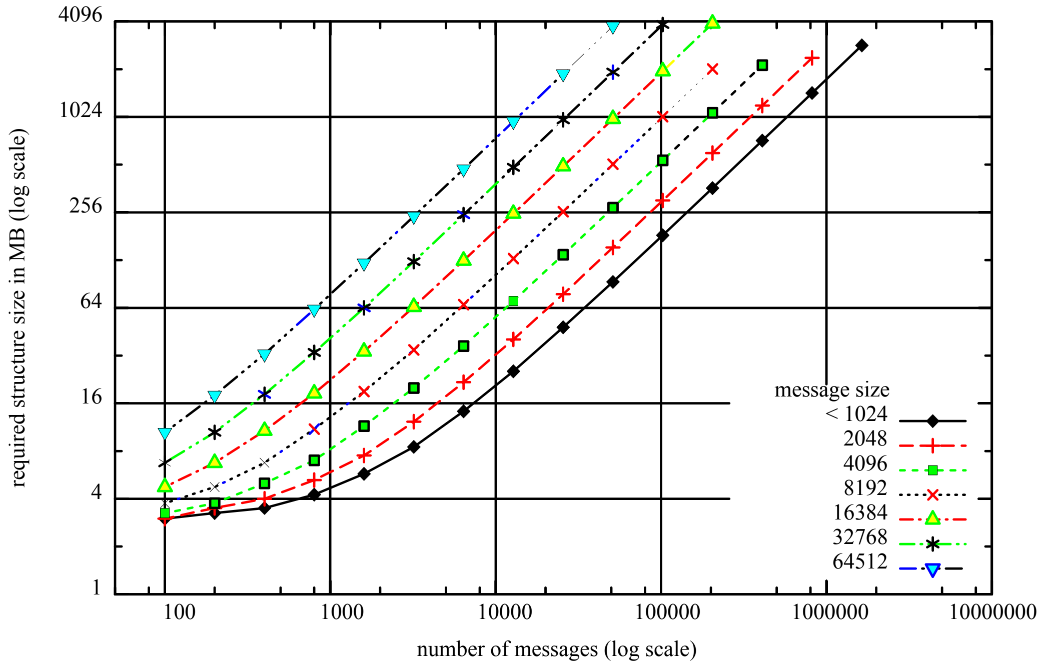
MQSeries CF Structure Allocation Size
OS/390 R9 CFCC Level 9



Note the log scales. For instance the tick marks between 1000 and 10000 on the x axis are the 2500, 5000, and 7500 messages points. Each tick mark up the y axis doubles the number of MB.

Here is the equivalent chart for a CF at CFCC level 10. Note that the CF list structure's own control storage size increases slightly.

MQSeries CF Application Structure Allocation Size
z/OS V1.2 CFCC Level 10



2.2.3 Increasing the maximum number of messages within a structure

The maximum number of messages can be increased dynamically either by

- Increasing the size of the structure within the currently defined limits or
- Changing the ENTRY to ELEMENT ratio

The ENTRY to ELEMENT ratio is initially fixed by WebSpere MQ. Every message requires an ENTRY and enough ELEMENTs to contain all message data and headers. This is why there is the same maximum number of messages for all sizes up to 1536 bytes including all headers. This ratio is not changeable under OS/390 V2.9 . System initiated alter processing, available from OS/390 V2.10 can adjust the ENTRY to ELEMENT ratio dynamically according to actual usage. It can also change the size of a CF list structure up to the maximum as defined for that structure.

2.2.3.1 *Use of system initiated alter processing*

This facility should only be used for Websphere MQ shared queues if the fix for the operating system APAR OW50397 is applied.
From OS/390 V2.10 the following CF list structure definition is possible

```
STRUCTURE NAME(PRF2APPLICATION1)
          SIZE(1000000)          /* size can be increased by OS/390*/
          INITSIZE(500000)       /* from 500000K to 1000000K by    */
          ALLOWAUTOALT(YES)   /* system initiated ALTER processing */
          FULLTHRESHOLD(80)
          PREFLIST(S0CF01)
```

When the FULLTHRESHOLD is crossed the operating system will take steps to make adjustments to the list structure ENTRY to ELEMENT ratio to allow more messages to be held within the current size, if possible. It will also, if necessary, increase the size towards the maximum (the value of SIZE). This process is not disruptive to ongoing work. However, it can take up to several minutes after the threshold is crossed before any action is taken. This means that a structure full condition, MQSeries return code 2192, could easily occur before any such action is taken.
For message sizes less than 956 bytes (5 * 256 - length(MQMD)) considerably more messages can be accommodated in the same size structure after any such adjustment.

2.2.3.2 *User initiated alter processing*

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

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

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

Decreasing CF structure size is not recommended as there are circumstances where it is very disruptive to performance for a considerable time.

2.2.4 How often should CF structures be backed up?

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

If backup is infrequent then recovery time could be very long and involve reading many active and archive logs back to the time of last backup. Alternatively an application structure can be recovered to empty with a RECOVER CFSTRUCT(...) TYPE(PURGE) command.

The time to achieve a recovery is highly dependent on workload characteristics and the DASD performance for the log data sets of individual systems. However, you can probably aim to do backups at intervals greater than or equal to the desired recovery time. CF application structure fuzzy backups are written to the log of the queue manager on which the BACKUP command is issued.

The overhead to do a backup is often not significant as the number of messages in an application structure is often not large. The overhead to do a backup of 100,000 1KB persistent messages is less than 2 CPU seconds on a 9672-X27 system.
The recovery processing time is made up of the time to

- restore the fuzzy backup of the CF structure, which is typically seconds rather than minutes.
- reapply the net CF structure changes by replaying all log data written since the last fuzzy backup.

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

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

2.2.4.1 Backup frequency example calculation

Consider a requirement to achieve a recovery processing time of say 30 minutes, excluding any reaction to problem time.
As an example, using ESS 2105-E20 DASD with the queue manager doing backup and restore on a 9672-X27 system running z/OS V1R2, we can restore 100,000 1KB persistent messages from a fuzzy backup on an active log in 20 seconds.

So, to meet the recovery processing target time of 30 minutes, we have more than 29 minutes to replay the log with the most data written since the last fuzzy backup. The maximum rate at which we can read a log data set backwards is about 3MB/sec on this system, so we can read about 5220MB of the longest log in 29 minutes. The following table shows the estimated backup interval required on this example system for a range of message rates:

1KB persistent msgs/sec to longest log	*1KB persistent msgs/sec to 3 evenly loaded logs*	*MB/sec to longest log*	*Backup interval in minutes (= time to write 5220 MB of data to longest log)*
400	1,200	0.97	90
1,200	3,600	2.9	30
2,200*	6,600	5.3	16
(* 2,200 is the maximum for this DASD with 1KB msgs)			

A crude estimate for the amount of log data per message processed (put and got) by queue managers in a QSG is message length plus 1.5KB.

2.2.5 Is DB2 tuning important ?

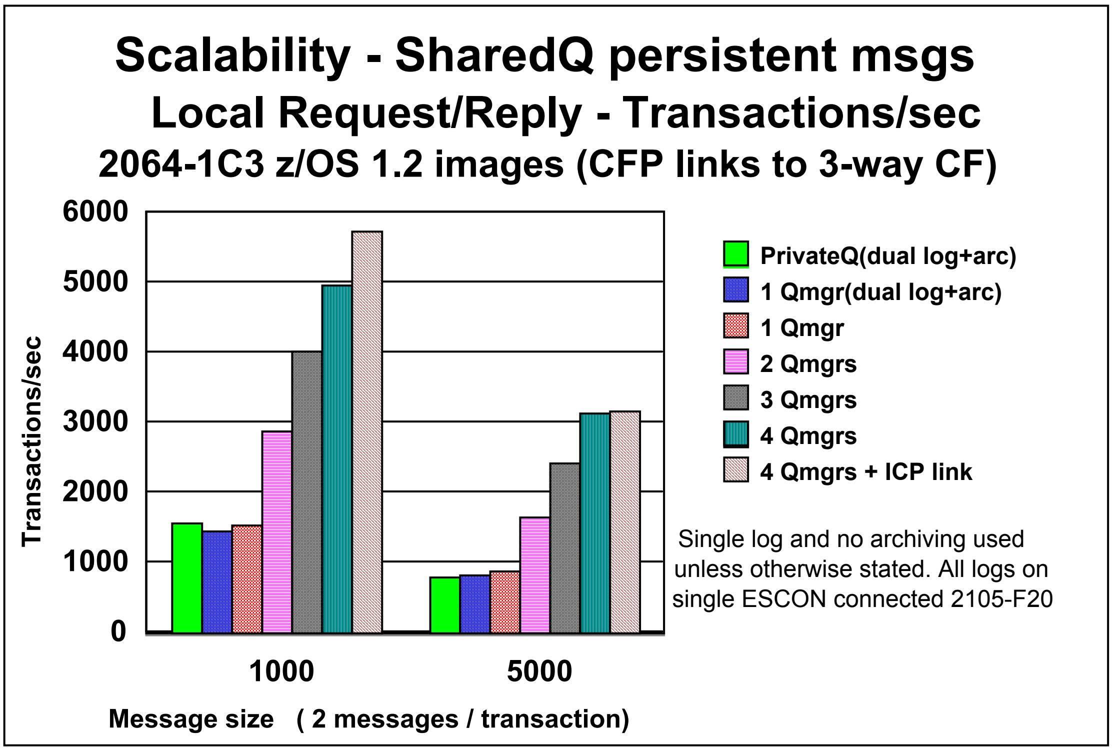
Yes, because DB2 is used as a shared repository for both definitional data and shared channel status information. In particular BUFFERPOOL and GROUPBUFFERPOOL sizes need to be sufficiently large to avoid unnecessary IO to DB2 data and indexes at such times as queue open and close and channel start and stop.
The DB2 RUNSTATS utility should be run after significant QSGDISP(SHARED) or QSGDISP(GROUP) definitional activity, for instance, when first moving into production. The plans should then be re-bound using SCSQPROC(CSQ45BPL). This will enable DB2 to optimize the SQL calls made on it by the queue manager.

2.3 Shared queue persistent message performance

2.3.1 Shared queue persistent message - throughput

We can process more than 11000 persistent messages per second with four queue managers using a pair of shared queues, as seen in the following chart and explanation.

Persistent message throughput is limited by the rate at which data can be written to the log. Having multiple queue managers each with its own log allows a many times increase in the throughput. Using shared queues means that this many times increase is available through a single queue or set of queues.



These results were obtained on a parallel sysplex LPARed out of one Freeway 2064-116 box with a single ESCON connected Shark ESS 2105-F20 DASD subsystem. This single Shark was not enough to run four queue managers each with dual logs and dual archiving at these rates. Real production parallel sysplexes would need, and might reasonably be expected to have, sufficient log DASD I/O rate capacity for the required number of logs and archives.

'Local Request/Reply' is a set of identical request applications and a set of identical reply applications running on each queue manager such that requesters MQPUT a message to a common server shared queue and MQGET their specific reply message from a common reply shared queue which is indexed by MsgId. The reply applications MQGET the next request message from the common queue, MQPUT the specific reply message to the indexed shared queue and MQCMIT. Thus there are two messages completely processed (that is created with MQPUT and consumed with MQGET) for each request/reply transaction.

The above chart demonstrates the following

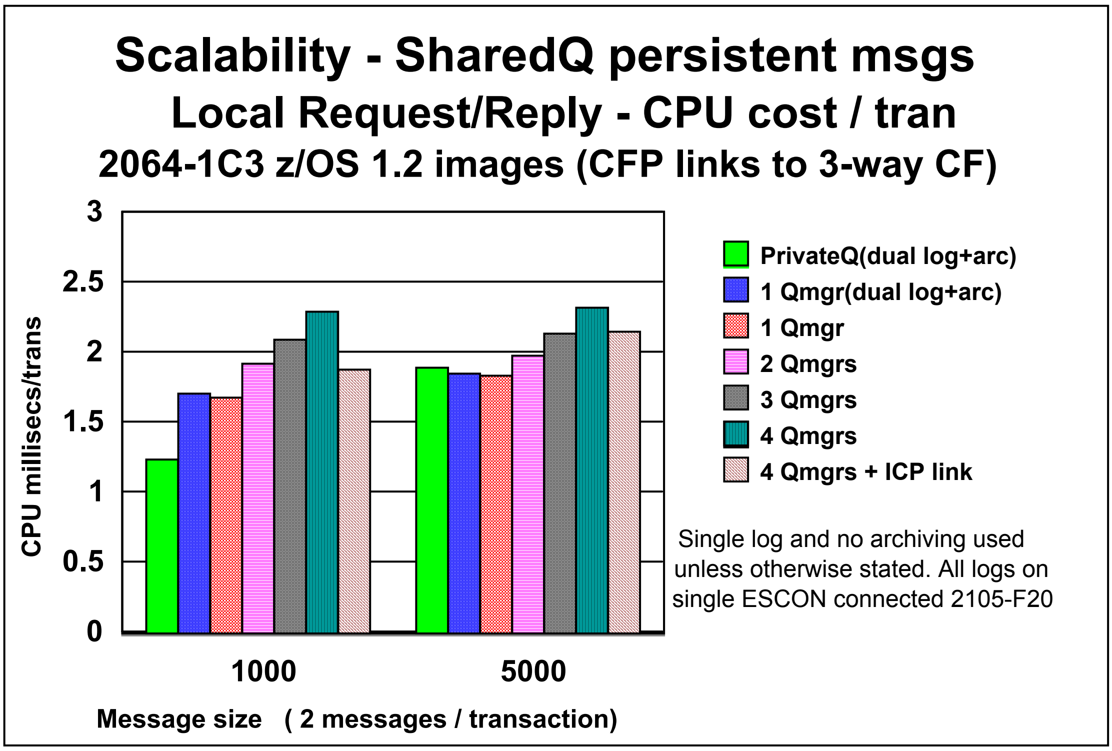
- More than 11000 messages per second(two messages per transaction) with 1000 byte messages.
- More than 6000 messages per second with 5000 byte messages.
- Single queue manager throughput for shared queues is of order 8% less than for private queues.

- dual log plus archiving versus single log does not have a huge effect on throughput (1 Qmgr(dual log+arc) versus 1 Qmgr).
- Scalability for 1 to 4 queue managers, where there is sufficient log DASD capacity and sufficient CF processing power.
- The potentially significant effect of the type of CF links used (4 Qmgrs versus 4 Qmgrs + ICP link). See CF link type and shared queue performance on page 0 .

2.3.2 Shared queue persistent message - CPU costs

CPU costs for shared queue are more than usually difficult to estimate from measurements of individual MQSeries operations. These measurements are probably the most representative of likely costs for real workloads. This is because they include the interactions between the CF and the queue managers in real life queue sharing under load including particularly waiting MQGET situations.

These CPU millisecs are derived from RMF reported 'MVS BUSY TIME PERC' for each total system. Thus they include all system overheads and are larger than would be found from adding 'WORKLOAD ACTIVITY' reported CPU usage by address space.



Based on the above chart a rule of thumb may be derived that shared queue persistent CPU costs compared to 'best case' private local queue

- are of order 37% more than for 1000 byte persistent messages. Each extra queue manager increases the CPU cost by about 9%.

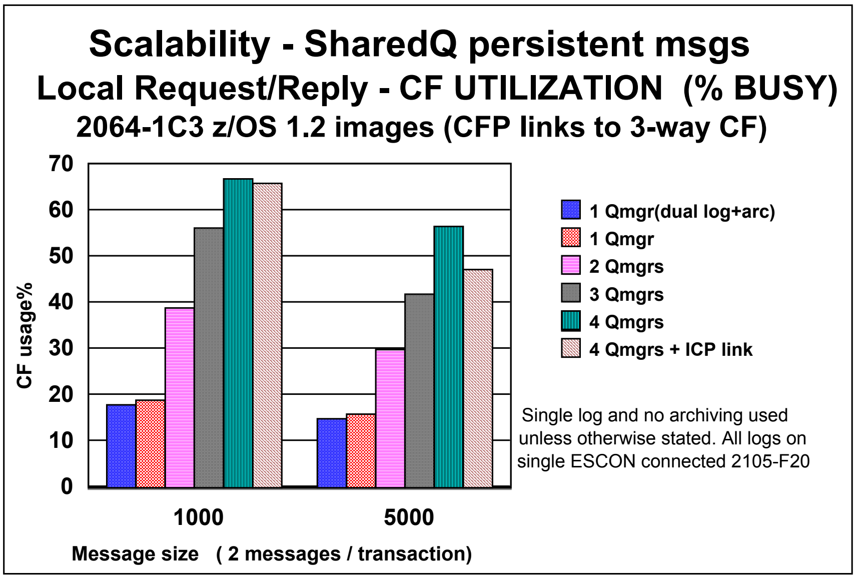
- Are similar for 5000 byte persistent messages. Each extra queue manager increases the CPU cost by about 8%.

'Best case' means all messages remain within the buffer pool. One of the advantages of shared queue is that there can be enough capacity to keep going even when there is an application instance outage. With private local queues any such outage could severely restrict throughput and/or cause buffer pools to fill with consequent more expensive performance until any backlog is cleared.

2.3.3 Shared queue persistent message - CF usage

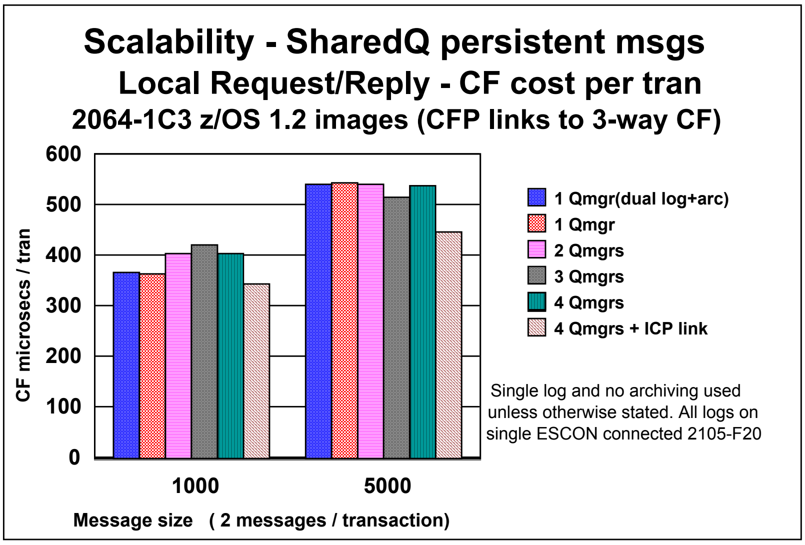
MQ calls to the CF are mainly to the application structures. There is no difference in the number and type of calls between persistent and non persistent messages. There are more calls to the CSQ_ADMIN structure for persistent messages.

The following chart shows the RMF reported 'CF UTILIZATION (% BUSY)' matching the persistent message local request/reply charts above.



A rule of thumb for CF cost is about 200 CF microsecs per message (400 CF microsecs per transaction) for 1000 byte messages when using CFP links.

For 5000 byte messages CF cost is about 270 CF microsecs per message when using CFP links.

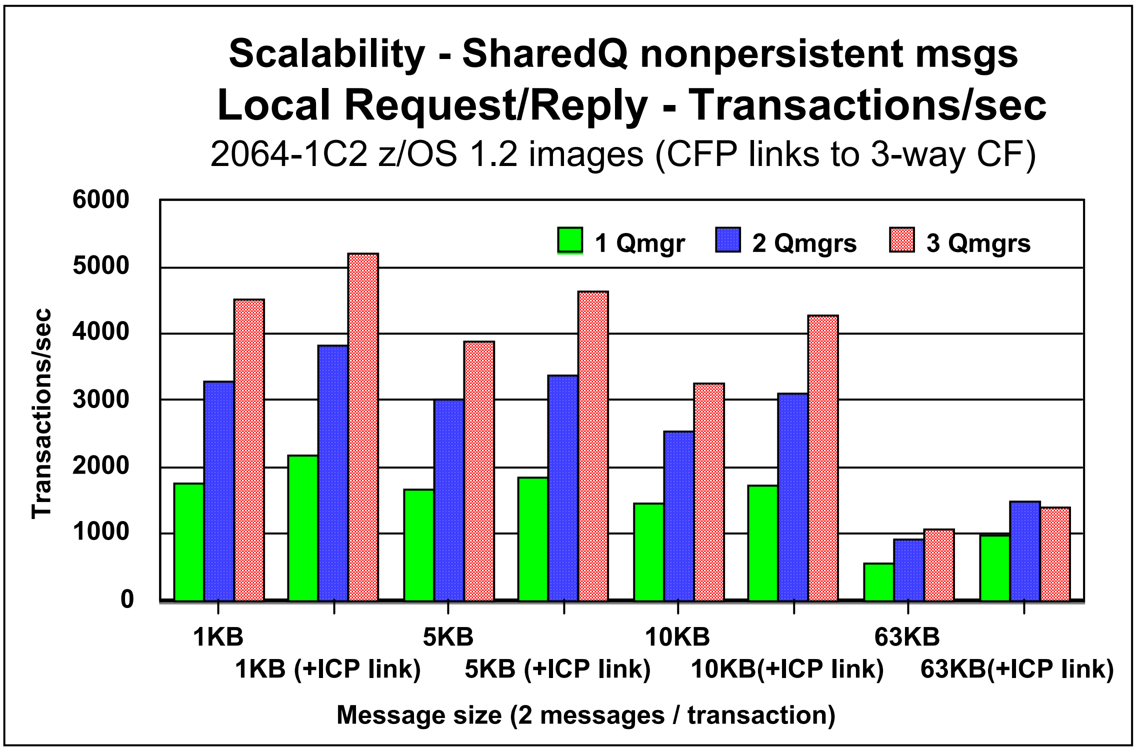


CF costs per message do not change significantly with increasing number of queue managers.

2.4 Shared queue non persistent message performance

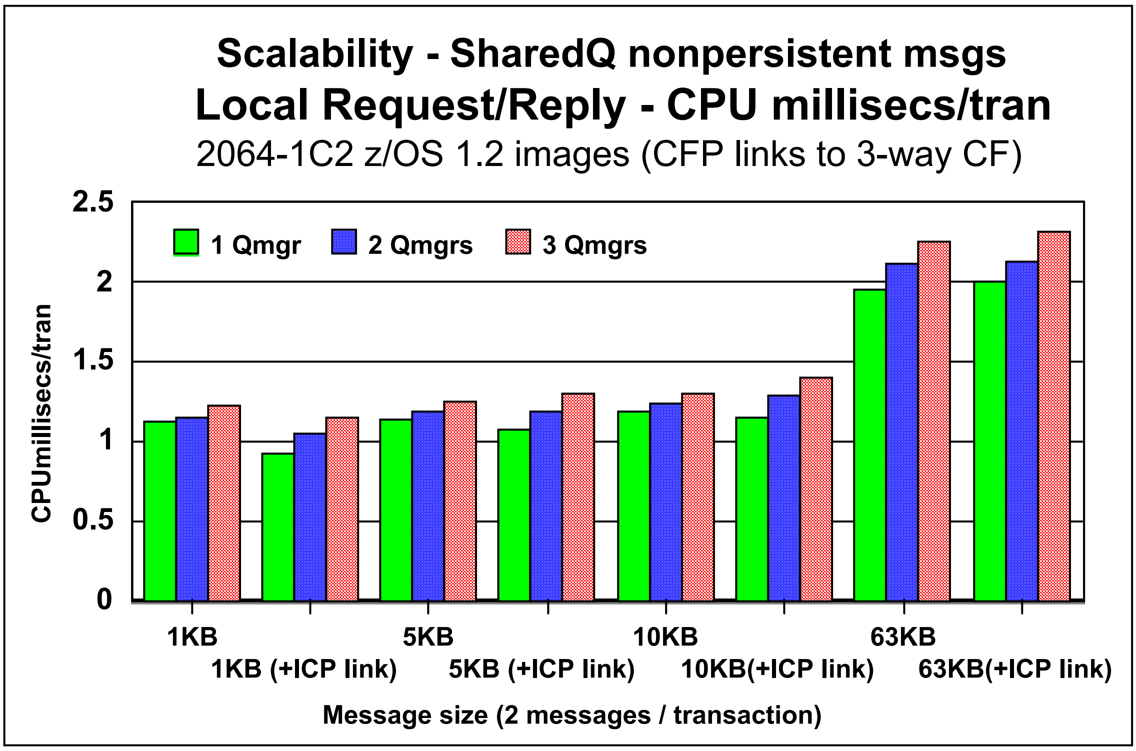
2.4.1 Shared queue non persistent message - throughput

Shared queue non persistent throughput is mainly limited by the power of the CF and the type of links to the CF.



Configuring one ICP link in addition to the four CFP links improves throughput and CPU cost. Previous experience using a 9672 and a Cluster Bus (CBS) link suggests that using a Cluster Bus (CBP) link might give throughput and CPU cost closer to that for ICP link than for CFP link.

2.4.2 Shared queue non persistent message - CPU costs



A rule of thumb may be derived from comparisons with results for private queue local request/reply (not shown here) that shared queue non persistent CPU costs compared to 'best case' private local queue

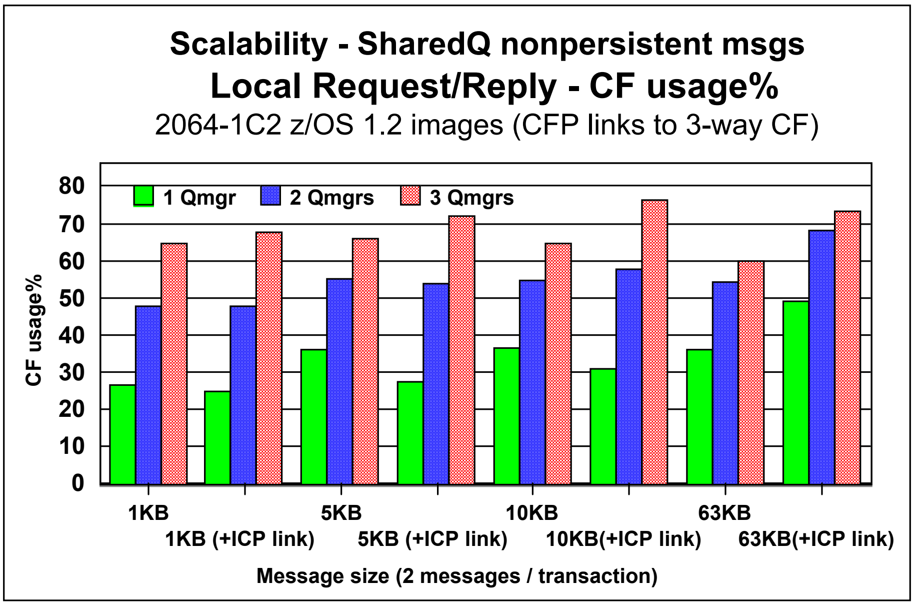
- Are of order 100% more than for 1000 byte non persistent messages.
- Are of order 75% more for 5000 byte non persistent messages.
- Then progressively down to 16% more for 63KB non persistent messages.
- Each extra queue manger increases the CPU cost by about 5%.

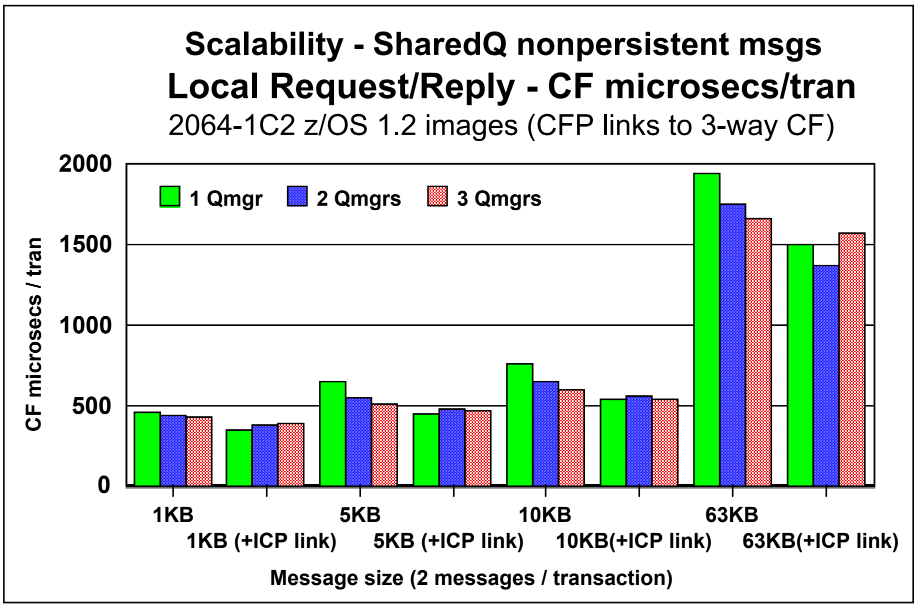
'Best case' means all messages remain within the buffer pool. One of the advantages of shared queue is that there can be enough capacity to keep going even when there is an application instance outage. With private local queues any such outage could severely restrict throughput and/or cause buffer pools to fill with consequent poorer more expensive performance until any backlog is cleared.

2.4.3 Shared queue non persistent message - CF costs

MQ calls to the CF are mainly to the application structures. There is no difference in the number and type of calls between persistent and non persistent messages. There are fewer calls to the CSQ_ADMIN structure for non persistent messages.

The following chart shows the RMF reported 'CF UTILIZATION (% BUSY)' matching the non persistent message local request/reply charts above.





A rule of thumb for CF cost is about 200 CF microsecs per message (400 CF microsecs per transaction) for 1000 byte non persistent messages when using CFP links.

For 5000 byte non persistent messages CF cost is about 270 CF microsecs per message when using CFP links.

2.5 Shared queue basic MQPUT,MQGET,MQCMIT CPU costs

In general the CPU time used for shared queue operations

- is more than for local queue operations particularly for short messages.
- is dependent on the type of CF link used, again particularly for short messages.

However, local queue operations are typically more expensive when messages have to be retrieved from page sets. This is likely to be necessary when messages exist for a long time and/or when buffer pool tuning is inadequate. Messages can exist for a relatively long time because of an application outage as well as by business application design. Shared queue message performance is not sensitive to the length of time for which messages exist as there is no buffer pool usage. Messages are always stored and retrieved directly from CF list structures.

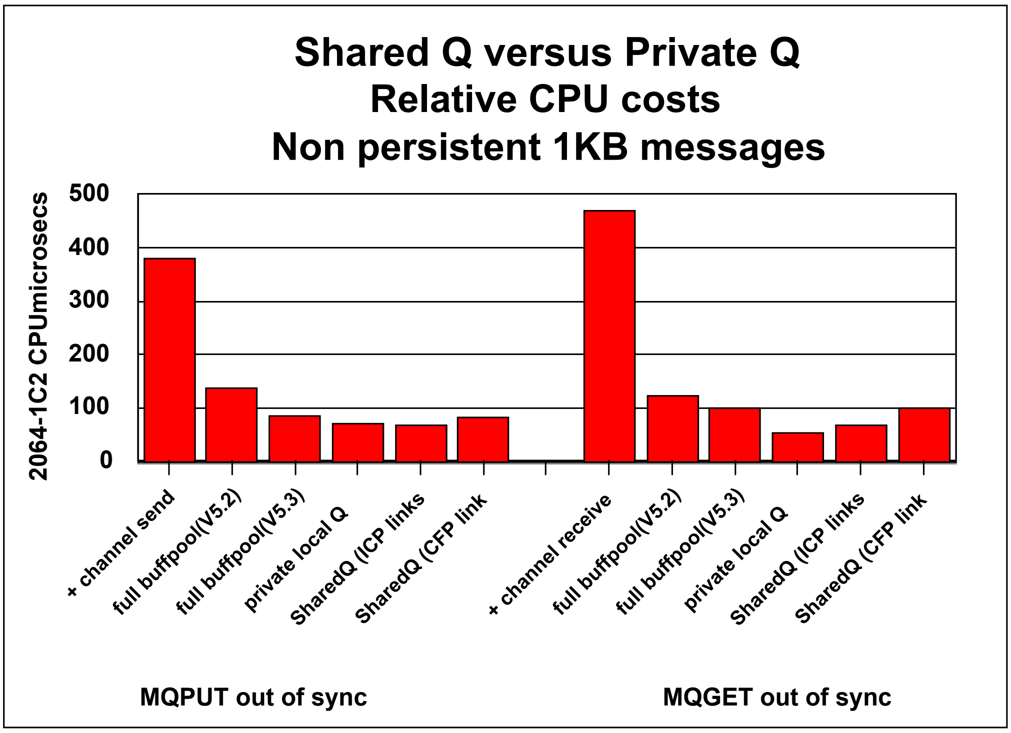
Remote queue operations are usually much more expensive than shared queue because of the extra costs involved in moving messages over channels. Where remote queues are currently within the same parallel sysplex they may be able to be replaced with shared queues with significant CPU cost saving.

Relatively expensive shared queue operations include MQOPEN/CLOSE (and hence MQPUT1), syncpointing (whether explicitly by MQCMIT or implicit by a transaction co-ordinator), and MQGET which has to wait. Relatively cheap operations include MQPUT out of syncpoint. Channel send and receive costs vary considerably depending on achieved batchsize on the channel.

CPU costs for shared queue operations are influenced by the type of links to the CF which are used and by heuristics in the operating system. See CF link type and shared queue performance on page 0. Therefore, it is not straightforward to predict any CPU cost difference when moving an existing application to the use of shared queues. However, the following charts give some indication of relative CPU costs for basic operations.

2.5.1 CPU costs for single 1KB non persistent messages - out of syncpoint

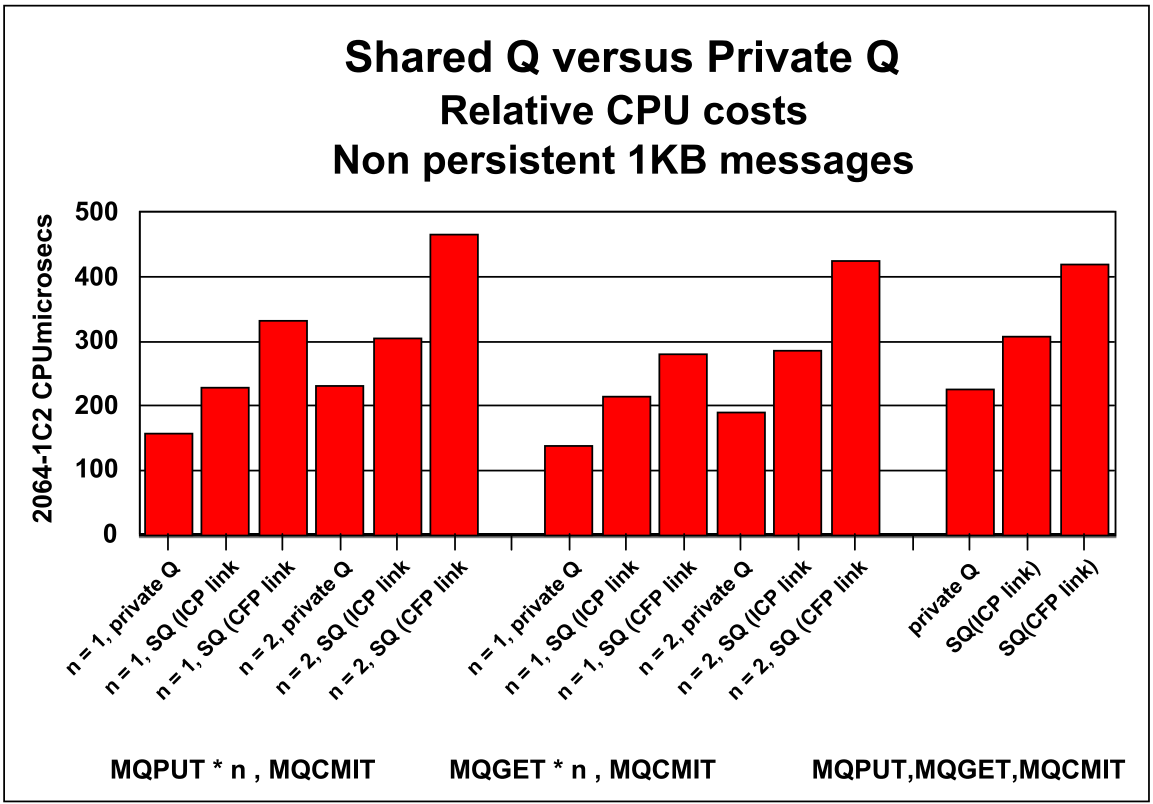
These CPU costs are TCB+SRB for a single threaded application plus queue manager, not derived from RMF %BUSY for a workload in a busy system.

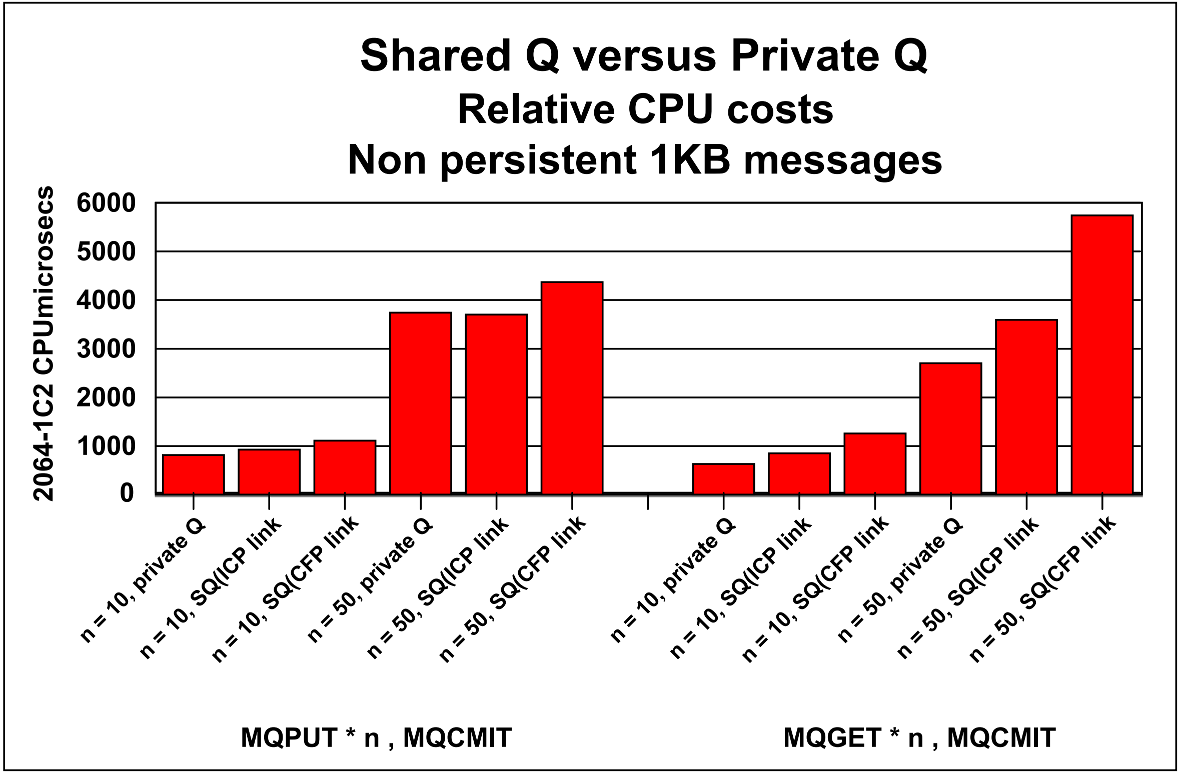


SQ - CPU costs for 1KB non persistent messages - within syncpoint

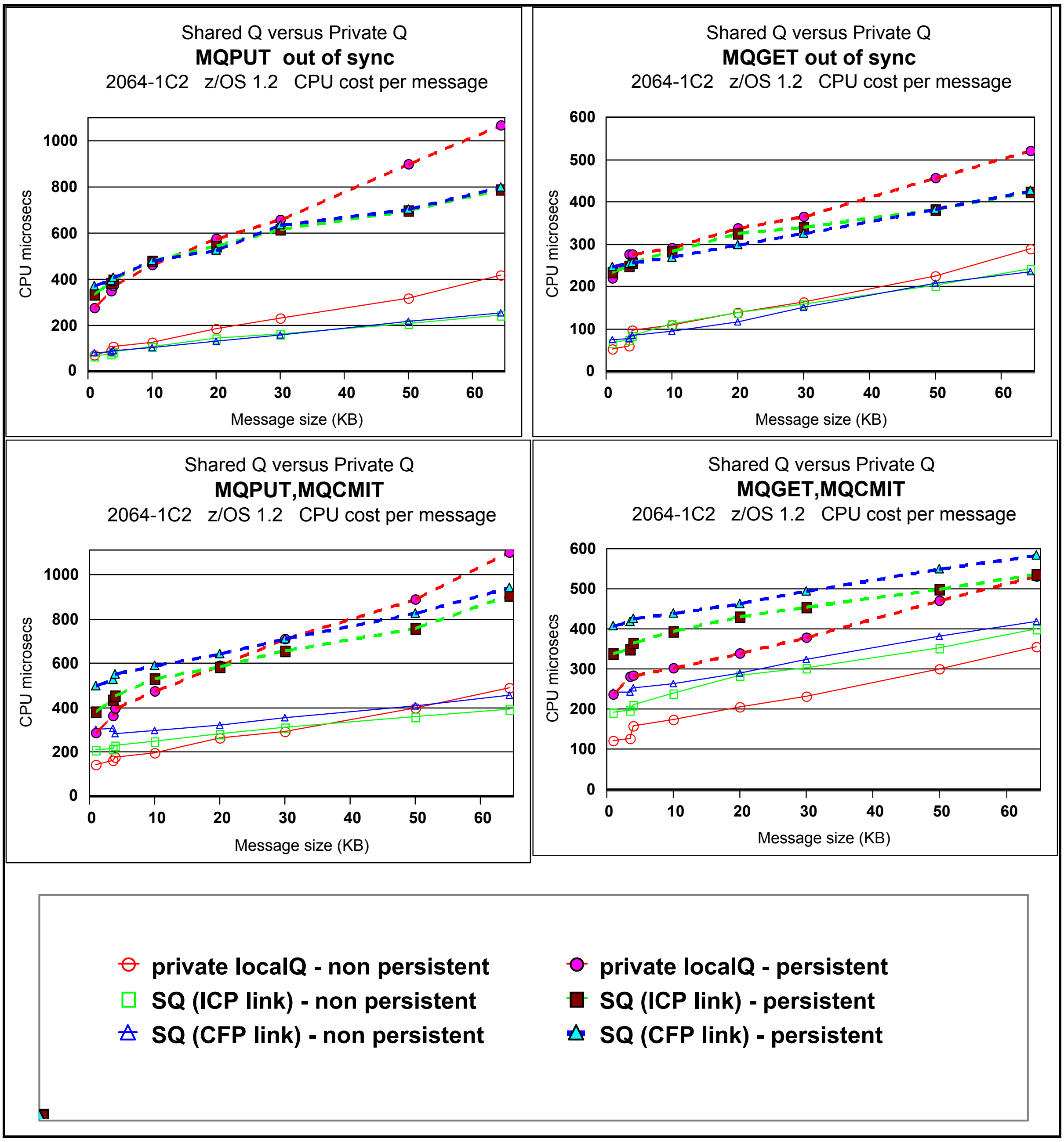
The next two charts show costs for

- a small number of operations within synchpoint including the MQCMIT.
- Larger numbers of operations within synchpoint including the MQCMIT.





2.5.2 SQ Persistent and non persistent CPU costs - by message size



2.6 Shared Queue performance improvements v V5.2

The CPU cost and number of CF accesses for MQPUT1 to a shared queue is substantially reduced.

The CPU cost and number of CF accesses for MQOPEN for input is also substantially reduced in many common usages. It is not reduced where complex triggering is specified for the queue. That is for trigger first with priority or trigger depth.

CPU time is reduced by about 20% for MQPUT1 and MQOPEN+MQCLOSE for input where there is no complex triggering.

3.0 Significant V5.2 performance PTF's incorporated

3.1 Persistent message throughput increase for dual logging

Persistent message throughput can be improved by up to 30% with this change for 1KB messages. The maximum log data rate and hence the maximum persistent message rate is not significantly changed.

Previous releases (except for V5.2 with PTF PQ54967) required some writes to the dual log to be done in series with the primary log. In this release (or for V5.2 with the PTF) all writes to dual logs are done in parallel if the primary log data set is on write cached DASD. With short messages and low throughput rates most writes to the secondary log data set would be in series. The QJSTSERW log statistic shows the number of serial writes, see SupportPac MP1B *Interpreting accounting and statistics data.*

3.2 Avoiding significant swapping of batch MQ applications

Batch applications, that is those which are link edited with CSQBSTUB or CSQRSTB or CSQRRSI, can have significant MVS swapping activity if there are MQGETs with wait or set_signal which actually have to wait. Significant extra CPU cost and reduction in throughput can be experienced.

To avoid this in previous releases either the applications must be made non-swapable or if running on V5.2 the PTF UQ56617 for APAR PQ50726 should be applied.

In this release (or V5.2 with the PTF applied) a swapable application will only be eligible for swap after a long wait (2 seconds). This is normal swapability for a batch application program so making such applications non-swapable with this release is not recommended.

4.0 Other V5.3 performance relevant changes

4.1 Other changes to logging

4.1.1 *Log record size increase for queue managers within a QSG*

To facilitate shared queue persistent message recovery across a QSG (queue sharing group) extra data is required in log headers for all log records of every queue manager in that QSG. This includes those for private local queue messages. This enables maximum persistent message rate within the QSG to scale across the number of queue managers, but reduces the absolute *maximum* rate of private local persistent messages a single queue manager can process. However, there is no change to log record sizes for a private queue manager (one not in a QSG).

Persistent message throughput is not significantly changed at typical (that is much less than the maximum) log data rates by this change. This is because throughput is most sensitive to the number of commits, which has not changed, rather than to these fairly small increases in the amount of log data.

Maximum log data rate is not typically approached except where there is a very heavy load of large messages. The reduction in private queue maximum persistent message rate is of order 1% for 4MB messages, 5% for 5000 byte messages, 7% for 1000 byte messages, and 10% for 100 byte messages. **Logs will consequently fill somewhat sooner than before and there will be an increase in archive log media requirement for a given workload of the order of the above percentages.**

A crude estimate of the amount of log data per persistent message is

- Message length plus 1.25KB for non QSG queue managers
- Message length plus 1.5KB for queue managers in a QSG

4.1.2 *Log data set pre-format*

Whenever a new log data set is created it must be formatted to ensure integrity of recovery. This is done automatically by the queue manager which uses formatting writes on first use of a log data set. This takes significantly longer than the usual writes. To avoid any consequent performance loss during first queue manager use of a log data set V5.3 supplies a log data set formatting utility.

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

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

The new log data set formatting utility made available with V5.3 (see SCSQPROC(CSQ4LFMT) job) can be used with new logs for previous releases.

4.2 Private indexed queue rebuild at restart

Private indexed queues have virtual storage indexes which must be rebuilt when a queue manager restarts. Prior to V5.3 these indexes are built sequentially queue by queue before initialization completes and thus before any application can be started. V5.3 can build these indexes in parallel and introduces the QINDXBLD(WAIT/NOWAIT) CSQ6SYSP parameter. WAIT gives previous release behaviour and is the default, NOWAIT allows initialization to complete before the index rebuilds complete.

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

Each index rebuild still requires that the first page of all the messages for that indexed queue be read from the page set. The elapsed time to do this is of the order of a few millisecs per page read. For instance, a private indexed queue consisting of 50000 messages of size 1KB which occupies 25000 page set pages increases elapsed time of restart by about 30 seconds in our system using Shark DASD. Buffer pool page usage is not significantly effected by V5.3 index rebuild. Thus other applications will not be impacted by buffer pool contention with index rebuilds.

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

There was an implementation limit on the maximum number of messages in a private indexed queue of about 1.8 million prior to V5.3. This limit is now about 13 million, which would require about 1GB of index virtual storage and so is not recommended!

QSGDISP(SHARED) indexed queues do not have queue manager virtual storage indexes.

4.3 Buffer pool usage improvements

4.3.1 Improved throughput when a buffer pool is too small

The throughput and CPU cost for messages which have to be written to page sets and later read back is significantly improved. We have seen CPU cost improvements of the order of 66% for non persistent messages of size 5KB and greater occupying 100,000 pages. For 1KB messages occupying 100,000 pages the CPU cost improvement we saw was 25%.

Buffer pools become too small when messages exist for an unexpectedly long time or when buffer pool tuning is inadequate or not possible because the amount of message data is too great. Messages can exist for a relatively long time because of an application outage. So this possible improvement in performance over previous releases may occur when most welcome. That is, immediately after an application is restarted when a large queue of messages has built up.

4.3.2 Up to 16 Buffer pools can be used

We recommend you remain with your existing, up to 4, buffer pool size and usage definitions unless you

- Have class of service provision reasons for separating one set of queues from another

- Have known queues which have different behaviours at different times or would otherwise be better isolated in their own individual buffer pools. This might be for their own performance benefit or to protect the other queues.

See *WebSphere MQ for z/OS Concepts and Planning Guide* for more information on buffer pool usage.

4.4 Maximum throughput through a single queue improved

For small messages

The maximum throughput possible through a single queue is improved by up to 25% for small non persistent messages because of improved internal latching. Most users wont see any significant benefit from this. But it may be a benefit in occasional extreme contention situations where previously MQPUTs would tend to outpace MQGETs with consequent queue length build up.

For large messages

The maximum throughput possible through a single queue is improved by up to 100% for large non persistent messages because of improved internal latching. The nature of the latching was such that significant contention was likely for concurrent operations for large messages on a queue. We have seen 100% improvement in throughput for 4MB non persistent messages with one MQPUT application in contention with one MQGET application both with no business logic. In real user applications with business logic any gains will be much smaller.

4.5 Checkpoint IO for non persistent messages reduced

There is usually much less IO to page sets than in previous releases for non persistent messages in private local queues at checkpoint. It is now possible that there is no such IO at all. This reduced IO load on the total system may be particularly welcome at checkpoints when an active log has just filled and archiving has commenced. It did not have a significant effect on our system where the total system IO load is not excessive other than to significantly reduce normal shutdown elapsed time.

4.6 CTHREAD usage

An anomaly whereby accounting trace costs could become very high if more than CTHREADS MQ threads are active has been removed in V5.3. CTHREADS now controls just the number of connections as intended and documented in the Systems Administration Guide.

An example of accounting threads exceeding connections is where there are hundreds of thin clients or message channels but only the value of CSQ6CHIP ADAPTERS parameter connections. If accounting trace is used in such cases, then in releases prior to V5.3, it is recommended that CTHREADS be set to 32760, its maximum value.

4.7 Faster log replay during recovery

The IO bound portions of recovery processing have been improved such that the elapsed time to recover after a queue manager failure (time between CSQR003I and CSQR006I console messages) can be reduced by about 25% on our system using Shark DASD (ESS 2105-E20).

4.8 Exit Chaining

This release of WebSphere MQ for z/OS introduces the facility to specify up to eight message exit names on a channel (previously only one message exit was permitted).

Message exit programs are loaded when the channel is started, and if a significant number of channels start in a short time a heavy IO requirement is generated to the exit libraries.

In this case the CSQXLIB concatenation should be included in the FREEZE(..) dataset lists to allow faster loading.

When starting 2000 channels, each with 3 message exits specified, our measurements indicated an improvement of approximately 15-20% in the time taken for the channel to start.

4.9 Costs of Moving Messages between MVS Images

This section looks at the total CPU costs of moving messages between queue managers in separate MVS images and compares those costs in WebSphere MQ for z/OS V5.3 with the previous release, V5.2. **There are no major changes in these costs.**

A driver application attached to a queue manager in System A puts a message to a remote queue which is defined on a queue manager running in System B. A server application in System B retrieves any messages which arrive on the local queue.

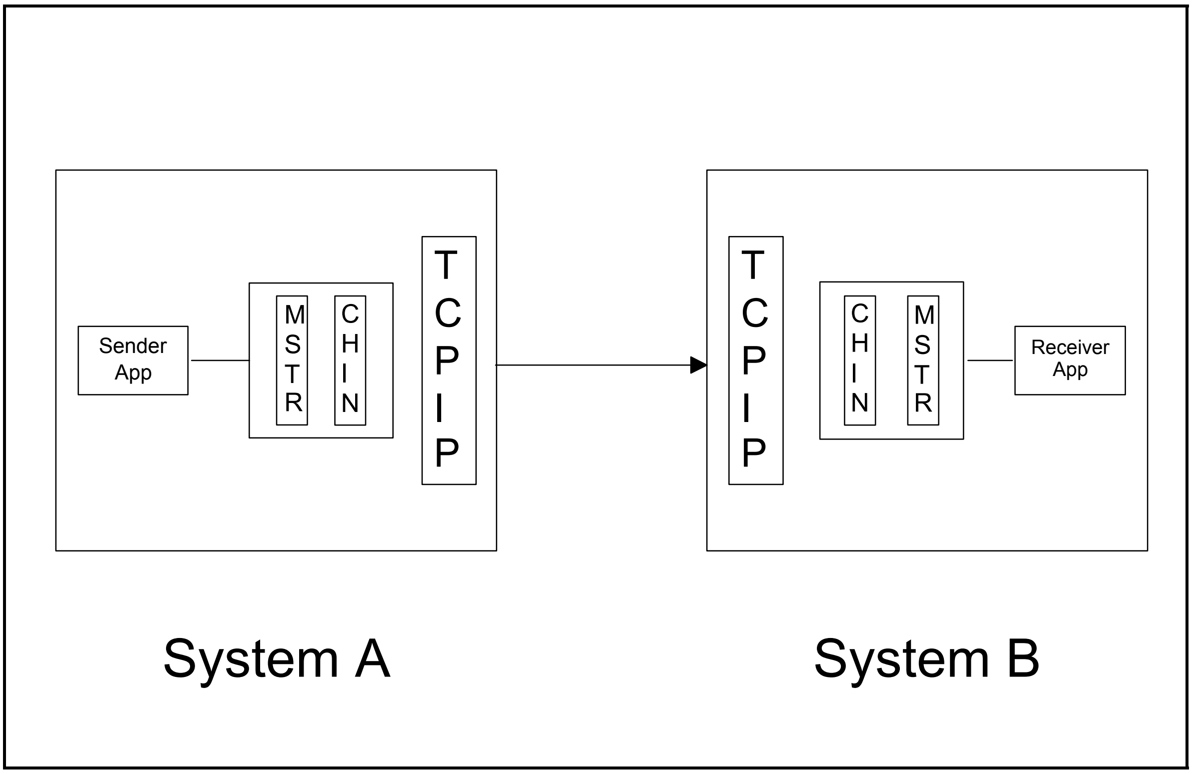


Figure: Set up for remote put/get measurements

The driver application continually loops, putting messages to the remote queue.
The server application continually loops, using get-with-wait to retrieve the messages.
Neither application involves any business logic.
The server application runs non-swappable.
The queue is not indexed.
The two MVS systems are connected via a 1GB Ethernet connection.

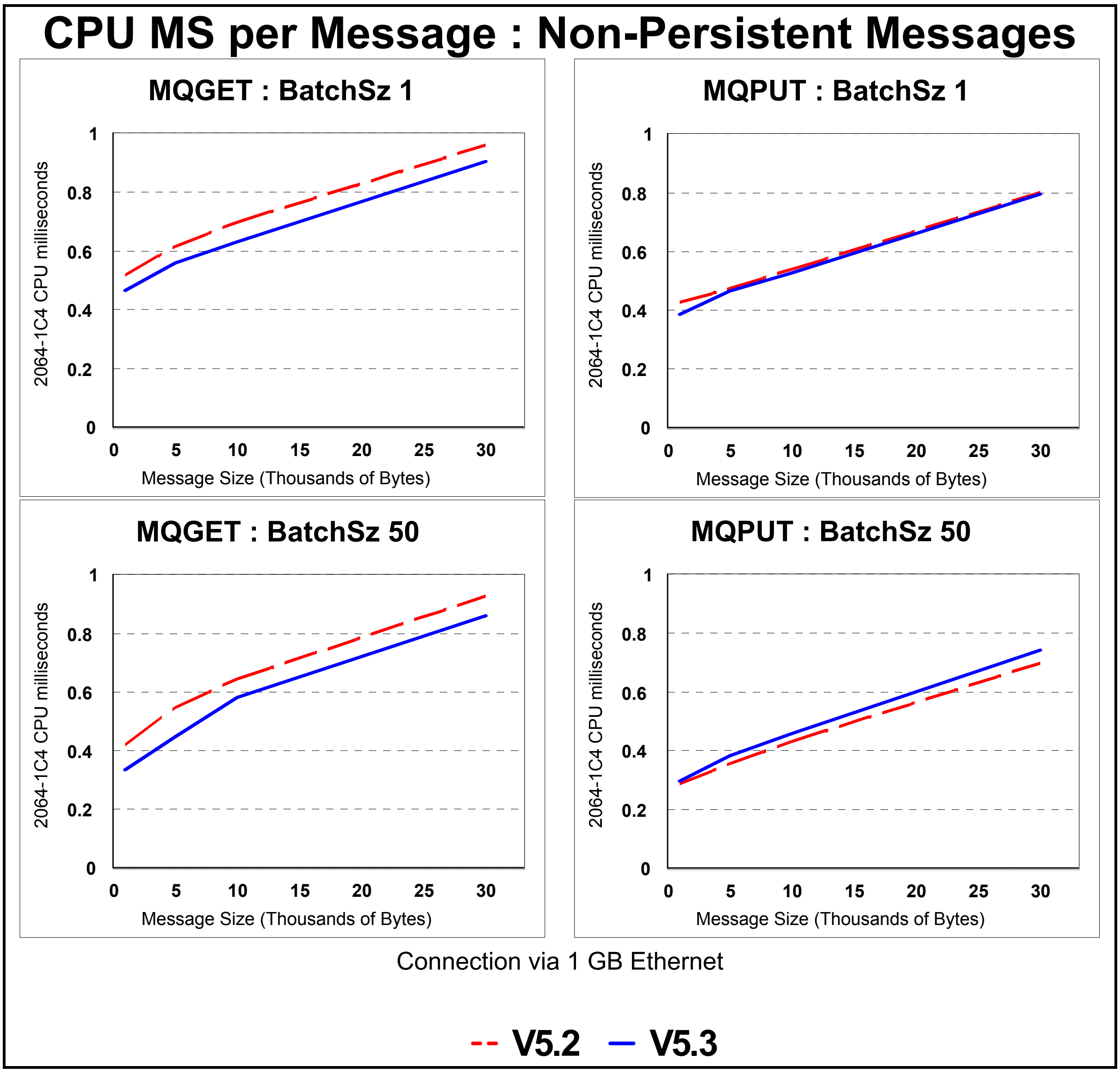
The measurements include all CPU costs for ‘Put and Send’ at the driver end and ‘Receive and Get’ at the server end.

4.9.1 Non-Persistent Messages

All non-persistent messages were put out of syncpoint by the sender application and got out of syncpoint by the server application. Measurements were made with two different channel settings:

- BATCHSZ(1) with BATCHINT(0) and NPMSPEED(FAST)
- BATCHSZ(50) with BATCHINT(1000) and NPMSPEED(FAST)

The charts below show the total CPU usage in both systems for non-persistent messages with a variety of message sizes.



Charts: CPU usage for Put+Send and Receive+Get - Non-Persistent Messages with Batch Sizes 1 and 50

The measurements for non-persistent messages indicate an improvement in the cost of getting messages irrespective of message size or batch size. The cost of putting messages is unchanged for a batch of 1, and shows a small increase for a batch size of 50.

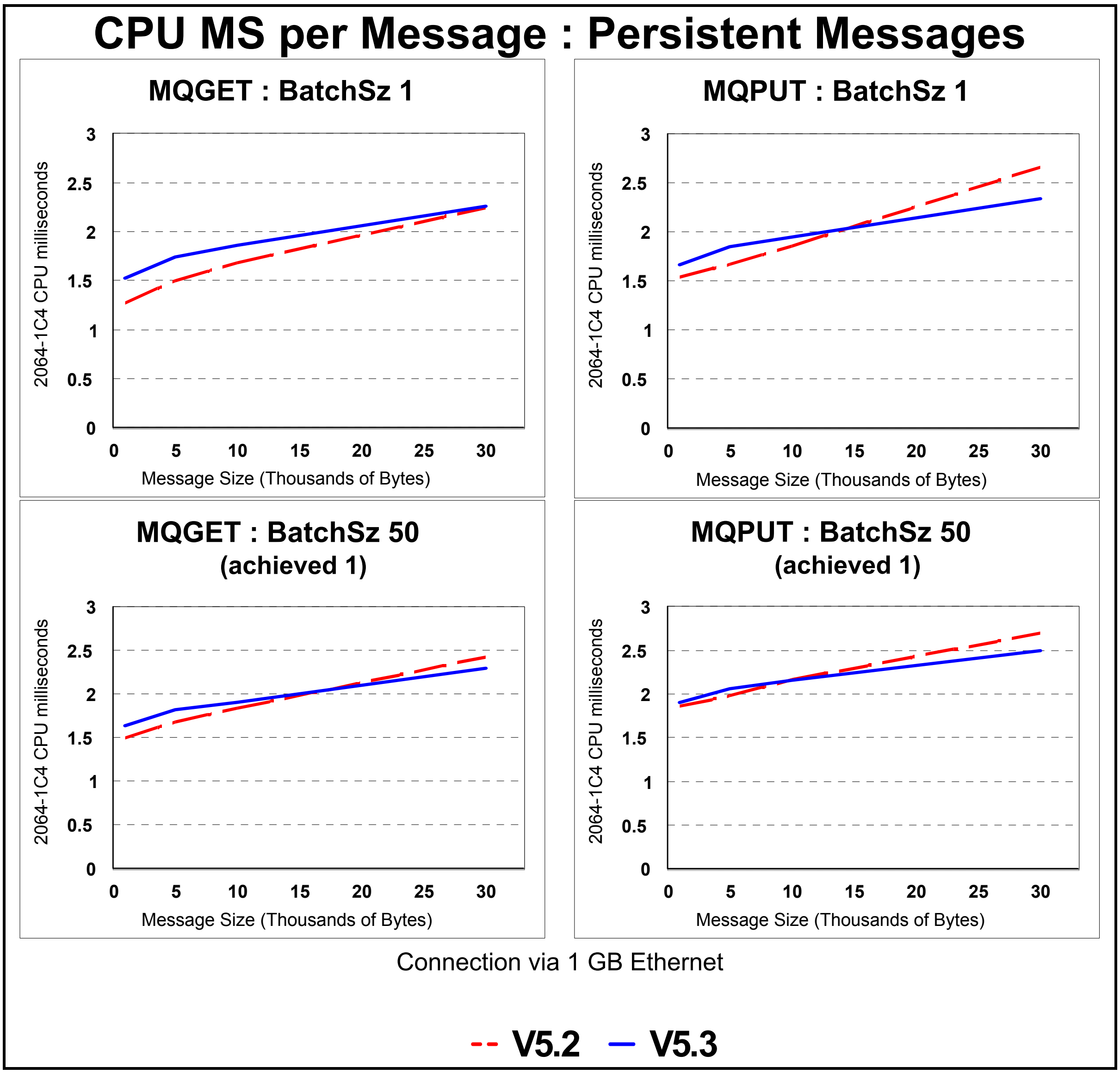
4.9.2 Persistent Messages

All persistent messages were put within syncpoint by the sender application and got within syncpoint by the server application. Measurements were made with two different configurations:

- BATCHSZ(1) with BATCHINT(0) and one message per driver commit. This represents the 'best choice scenario' for persistent messages where the expected message rate, and therefore the expected achievable batch size, is unknown.

- BATCHSZ(50) with BATCHINT(0), one message per driver commit and a brief (1/50th second) pause by the sender application after each put. This scenario represents a system using the default batch size and a low persistent message rate. Note that, since BATCHINT is set to zero, the *achieved* batch size in this scenario would be 1 even though BATCHSZ is set much higher.

The charts below show the total CPU usage in both systems for persistent messages with a variety of message sizes.



Charts: CPU usage for Put+Send and Receive+Get - Persistent Messages with BATCHSZ 1 and 50, (achieved batch size of 1 in all cases)

The measurements for persistent messages show a changed profile for both MQPUT and MQGET with batch sizes of 1 and 50. In all cases there is a trend towards lower CPU cost as the message size increases.

MQGET with a batch size of 1 shows an increase in CPU cost for all message sizes up to 30000 bytes. This situation is currently under investigation.

5.0 Secure Sockets Layer (SSL) Support

Secure Sockets Layer (SSL) support has been added to WebSphere MQ in this release.

SSL is an industry-standard protocol that provides a data security layer between application protocols and the communications layer. Using defined methods for data encryption, message integrity and client and server authentication, it is intended to provide protection against eavesdropping, tampering and impersonation.

SSL support in WebSphere MQ is implemented at the channel level and provides the following security services:

Mutual authentication

To guard against impersonation an initial SSL handshake occurs at channel start, during which the server provides a cryptographic certificate to prove its identity to the client. Optionally the client may also provide a certificate to the server

Message integrity

To defend against tampering a message hash function using a dynamic cryptographic key can be applied to messages which are moved across SSL channels. This ensures the detection of any changes to the content of a message as it travels between client and server.

Message privacy

To protect against eavesdropping, messages moving across SSL channels can be encrypted using a dynamic cryptographic key.

Specialised hardware, such as the Cryptographic Coprocessor Feature, is available to assist with the performance of certain cryptographic functions in a z/OS environment.

5.1 Channel Startup

Channels which use SSL perform a security handshake at startup. Starting an SSL channel pair consequently consumes more CPU than starting a non-SSL channel pair.

Two principal factors affect the performance of the SSL handshake in a z/OS environment:

- client authentication, which is optional and if selected increases the cost of starting a channel
- the availability of cryptographic hardware. Cryptographic hardware is not a prerequisite for SSL but it is recommended as its presence leads to a significant performance improvement at channel startup.

Note that the choice of SSL cipherspec (SSLCIPH) has little or no effect on the performance of the SSL handshake.

The authentication performed at channel start may optionally be extended to include Certificate Revocation List (CRL) checking using LDAP servers. This function, which establishes whether an otherwise valid certificate has subsequently been revoked, would be a potentially very significant overhead to the channel startup performance discussed in this section.

Client Authentication.

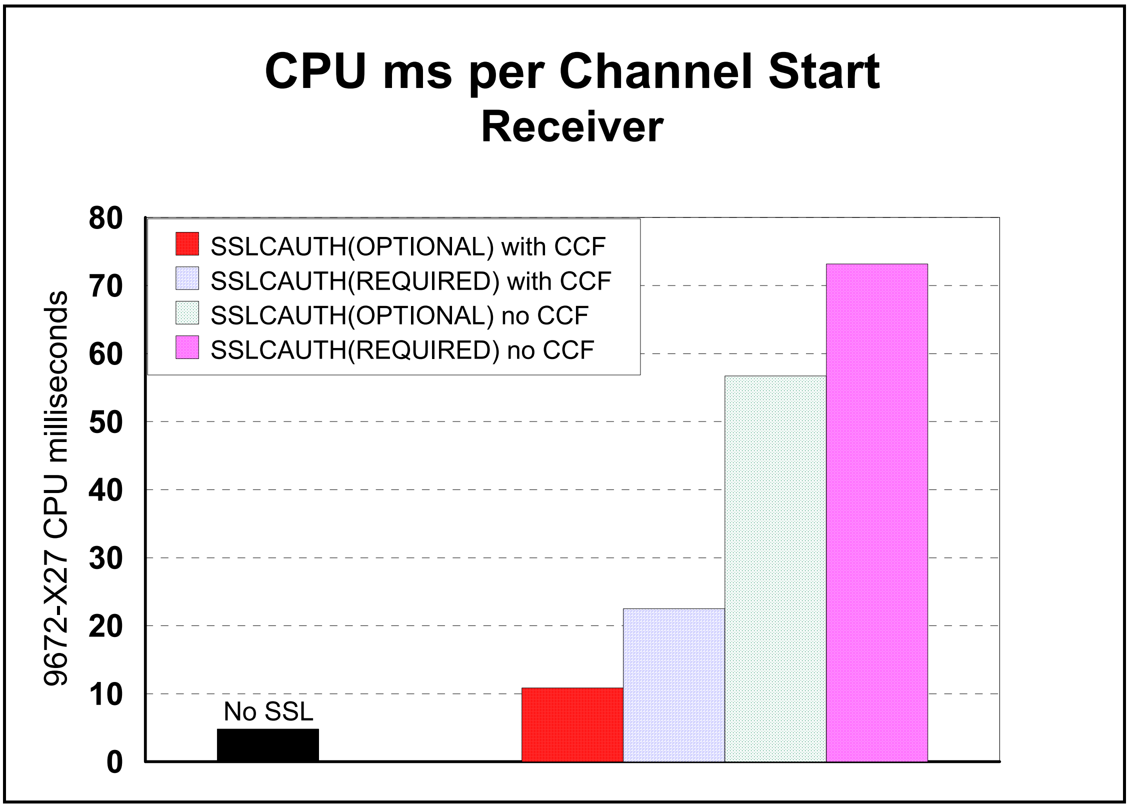
The initiating end of the channel (typically an MQ SENDER channel) acts as the SSL client; the other end of the channel, which receives the initiation flow (typically an MQ RECEIVER channel), acts as the SSL server. The SSL server is always required to provide a certificate to the client for authentication purposes. However, authentication of the SSL client may not be required. The SSLCAUTH parameter of the channel definition at the server determines whether client authentication is REQUIRED or OPTIONAL.

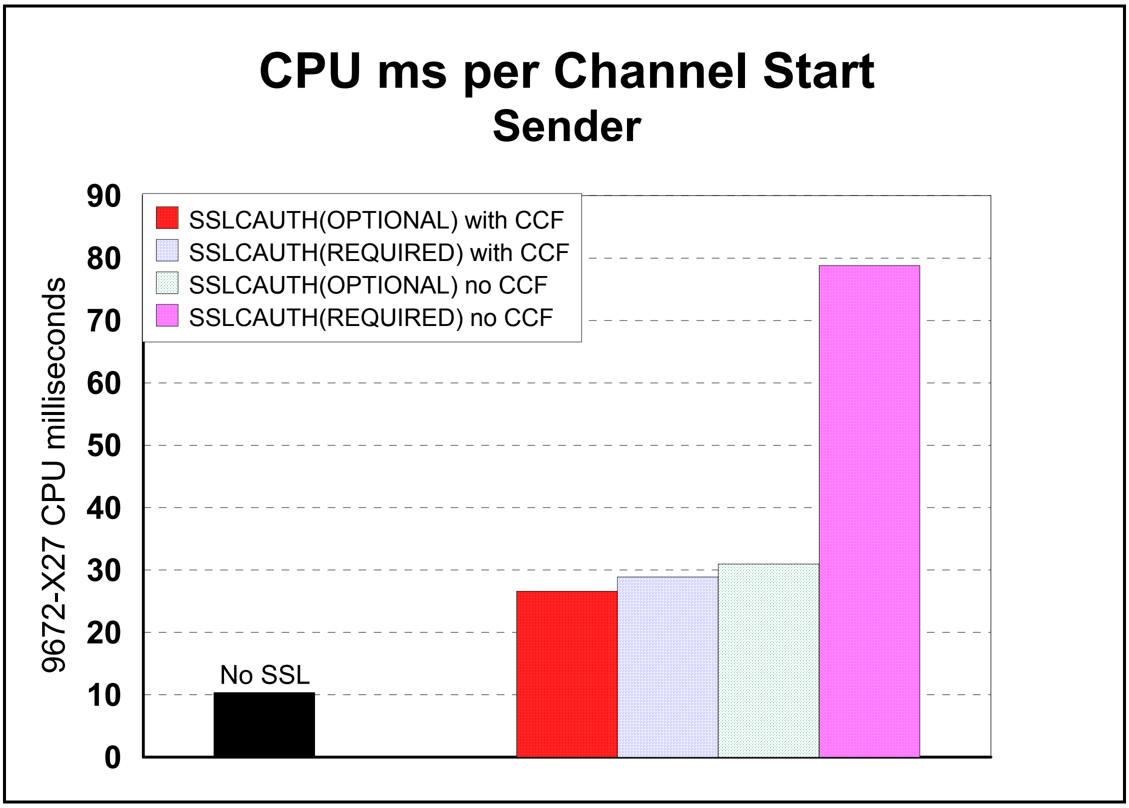
Note: The SSLCAUTH setting at the SSL server does not determine whether client authentication takes place; it determines whether authentication is *required*. If a certificate is available at the SSL client it will be sent to the SSL server and validated by the server whether it is required or not. For client authentication not to occur the SSL server channel must be set to SSLCAUTH(OPTIONAL) *and* there must not be a certificate available at the SSL client.

Cryptographic Hardware

Certain cryptographic function are faster when run on specialised hardware, such as the Cryptographic Coprocessor Feature. During the SSL handshake the Cryptographic Coprocessor Feature primarily assists with the retrieval of the local certificate. Thus the presence of a the availability of cryptographic hardware will always deliver a benefit at the receiver (SSL server) end because the SSL server is always required to provide a certificate.

The charts below illustrate the effects of client authentication and the value of the Cryptographic Coprocessor Feature at channel startup*. The Cryptographic Coprocessor Feature used for these measurements was the FRW1 complementary metal oxide semiconductor (CMOS) server, which is available as a feature on the S/390 Enterprise Servers and S/390 Multiprise.*





5.2 Moving Messages

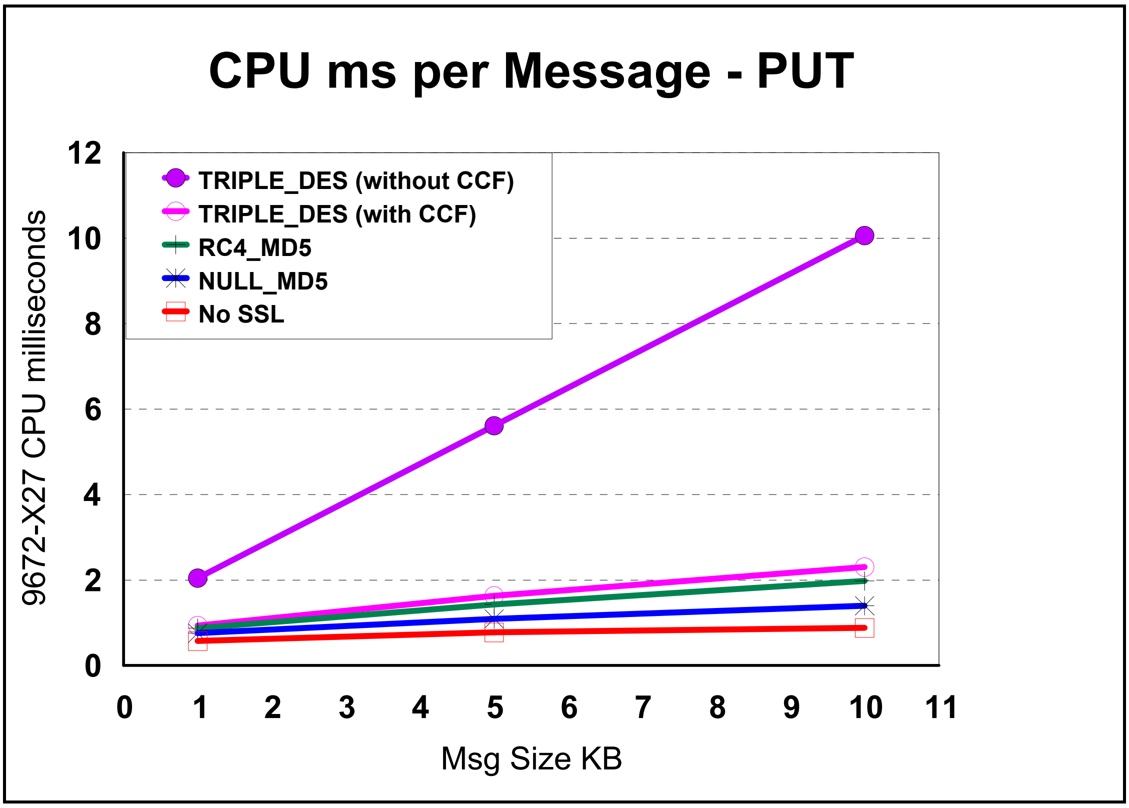
The SSLCIPH parameter determines the type and strength of encryption which is to be applied to messages moved across an SSL channel pair. The encryption and decryption of messages necessarily incurs a cost in terms of CPU consumption. This cost is determined by the size of the message, the type of encryption chosen and the availability of cryptographic hardware. Cryptographic hardware is essential for good performance when using the most powerful encryption algorithms.

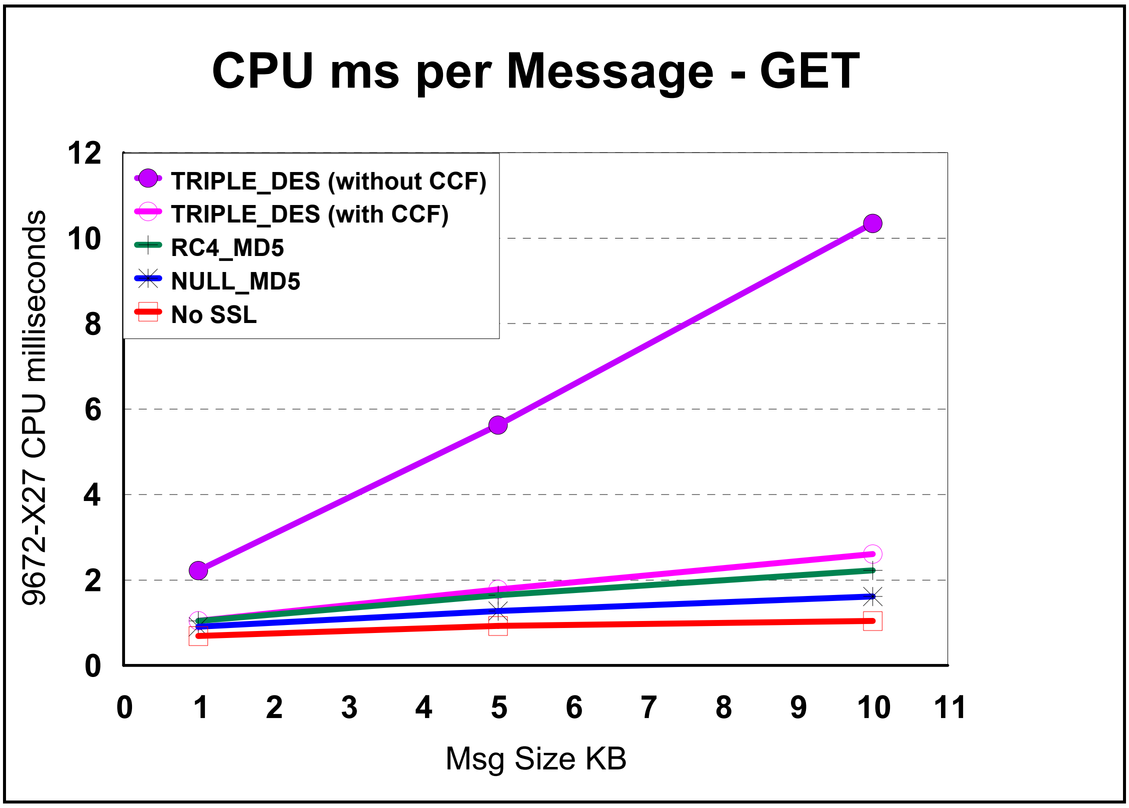
To demonstrate the CPU costs of message encryption we selected three different SSLCIPH settings:

• NULL_MD5	Protection against tampering but no encryption
• RC4_MD5_EXPORT	A typical medium strength encryption function
• TRIPLE_DES_SHA_US	The most powerful encryption function

The charts below demonstrate how the cost of encryption varies with message size. The costs shown here cover the total process of MQPUT, encrypt and send at one end of the transaction and receive, decrypt and MQGET at the other.

The Cryptographic Coprocessor Feature enables DES encryption functions to run more efficiently. Our measurements confirm that cipherspecs which do not involve the DES algorithm do not benefit from the availability of this feature. However, when using the TRIPLE_DES_SHA_US cipherspec the charts demonstrate that there is a substantial advantage to be gained in using the CCF.


CPU ms per Message - PUT
TRIPLE_DES (without CCF)
TRIPLE_DES (with CCF)
RC4_MD5
NULL_MD5
No SSL
9672-X27 CPU milliseconds
0
2
4
6
8
10
12
0
1
2
3
4
5
6
7
8
9
10
11
Msg Size KB


CPU ms per Message - GET
TRIPLE_DES (without CCF)
TRIPLE_DES (with CCF)
RC4_MD5
NULL_MD5
No SSL
9672-X27 CPU milliseconds
0
2
4
6
8
10
12
0
1
2
3
4
5
6
7
8
9
10
11
Msg Size KB

The cost of using SSL depends upon the cipherspec selected, but for all cipherspecs the additional CPU cost is approximately linear with message size. The table below shows the approximate factors which can be used to estimate the additional cost in CPU ms for a given cipherspec an message size.

Cipherspec	Action	Constant Factor	Coefficient per 1KB
NULL_MD5	Put + Send	0.15	0.045
	Receive + Get	0.12	0.49
RC4_MD5_EXPORT	Put + Send	0.15	0.1
	Receive + Get	0.2	0.105
TRIPLE_DES_SHA_US with CCF	Put + Send	0.2	0.14
	Receive + Get	0.2	0.144
TRIPLE_DES_SHA_US without CCF	Put + Send	0.5	0.89
	Receive + Get	0.5	0.9

Examples:

For a 5KB message using RC4_MD5_EXPORT,
additional CPU at the SENDER = 0.15 + (0.1 * 5) = 0.65 CPU ms
additional cost at the RECEIVER = 0.2 + (0.105 * 5) = 0.725 CPU ms

5.3 What is the maximum number of SSL channels?

Every SSL channel uses approximately 210 KB of virtual storage in the extended private region (EPVT) (+ message length for messages of greater than 32KB).

This means that the maximum number of SSL channels which can run concurrently is likely to be around 7500 on most systems, where EPVT size is unlikely to exceed 1.6GB.

The maximum number of SSL channels where maximum message size is greater than 32KB is of order

EPVTsize(KB) / (max-message-size(KB) + 210)

So, for example, if your started channels are all required to move 4000KB messages and we assume the EPVT size is 1600000 KB then only about 380 channels are possible. If all channels are to move 100 MB messages then only 15 channels are possible

5.4 What value should be used for SSLTASKS?

Server Tasks, similar to the adapter tasks already in the Channel Initiator address space, are required to run the SSL Handshake and encryption calls. The number of these tasks started is specified using ALTER QMGR SSLTASKS.

We have seen no benefit from using more than 8 as the value of the SSLTASKS attribute in an environment where there is no Certificate Revocation List (CRL) checking.

If CRL checking is used then an SSLTASK is held by the channel concerned for the duration of that check. This could be for a significant elapsed time while the relevant LDAP server is contacted. Each SSLTASK is a z/OS TCB.

6.0 Message Grouping

Message grouping, which is already available in WebSphere MQ on certain other platforms, has been added to WebSphere MQ for z/OS in this release. For private queues this feature is provided at little additional cost.

Message grouping offers a number of features which may lead to improved application performance:

- the optional ability to MQGET the first message of a group only if that group is complete
- The ability to MQGET messages in the intended logical order rather than the physical order in which they appear in the queue
- all messages in a group are handled by the same application, allowing different applications to work concurrently on different groups of messages

To demonstrate the costs of message grouping we compared two similar scenarios. In the first scenario a number of (non-grouped) messages were added to a queue as a single unit of work, that is all messages were put within syncpoint and then committed, and then the batch of messages was retrieved in the same way. In the second scenario the same number of messages was put and retrieved as a message group.

These two scenarios were repeated for a range of different group sizes and message sizes.

6.1 *Message Grouping with Private Queues*

The charts below show the results of these measurements in terms of the *average CPU cost per message* for non-persistent messages using a private (i.e. non-QSG) queue.

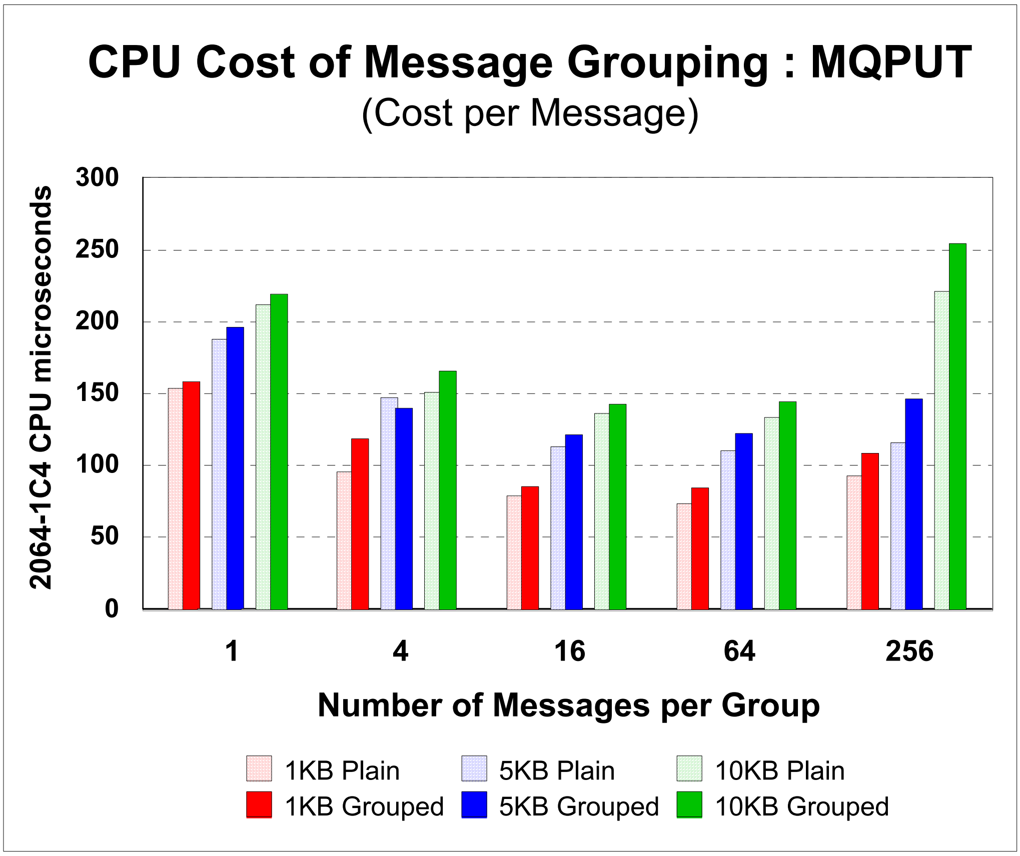


Chart: CPU Cost per Message for MQPUT with Grouping - Non-persistent Messages, Private Queue

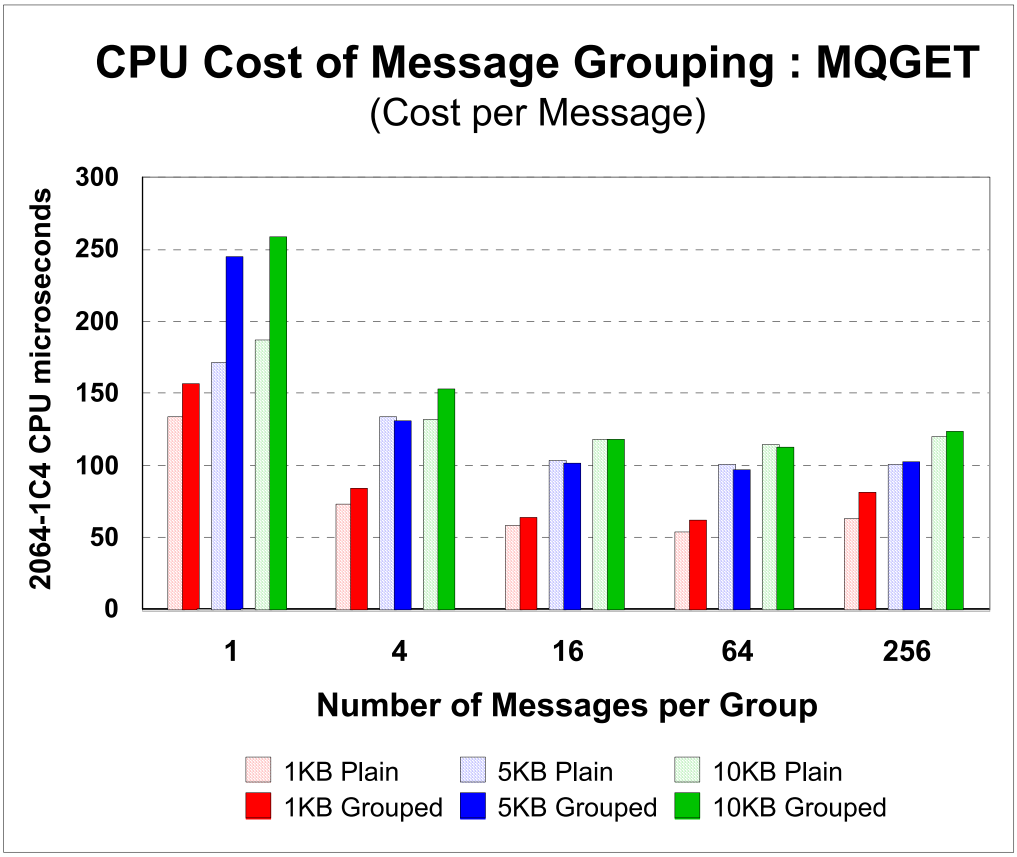


Chart: CPU Cost per Message for MQGET with Grouping - Non-persistent Messages, Private Queue

Using message grouping typically introduces a small increase in CPU consumption for both MQPUT and MQGET. The CPU overhead, which varies with the number of messages in the group, should be set against the benefits of improved application performance.

Note that the largest overheads are for getting a single message group - that is a single message which is identified as both the first and the last message in the group. This would not be regarded as a normal occurrence for message grouping.

6.2 Message Grouping with Shared Queues

The charts for shared queues show little or no additional cost when putting messages to the queue, and a consistent CPU overhead when retrieving messages in groups.

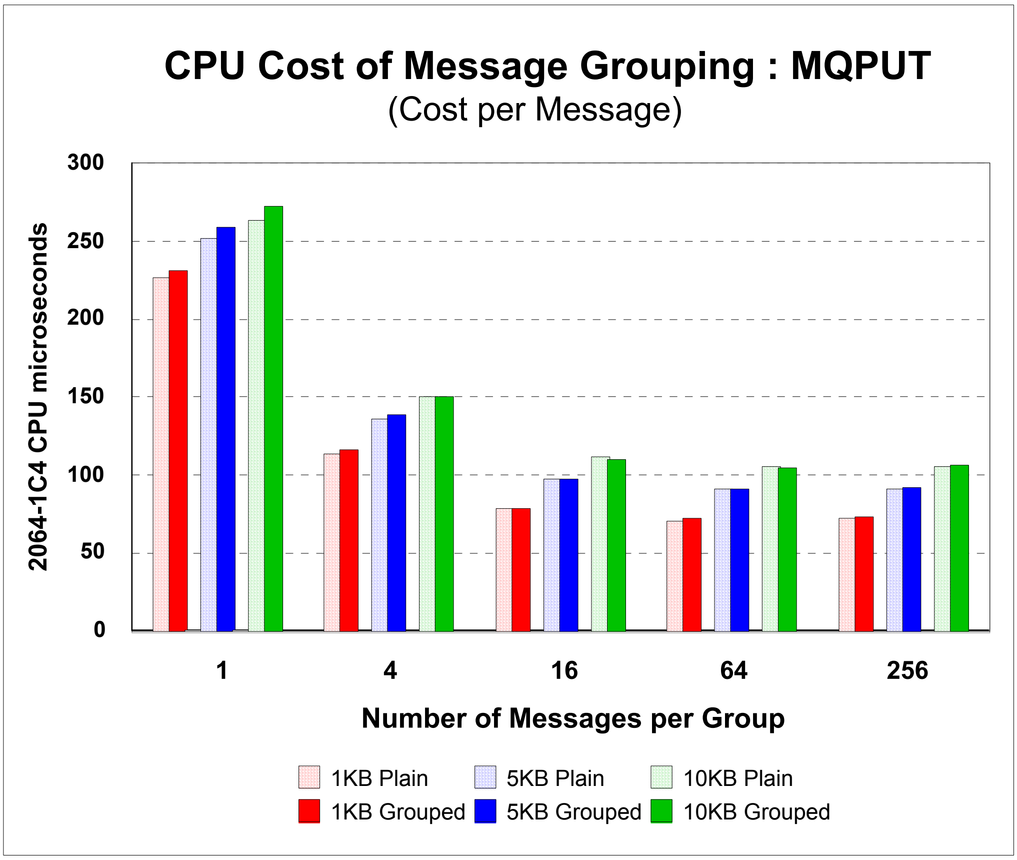


Chart: CPU Cost per Message for MQPUT with Grouping - Non-persistent Messages, Shared Queue

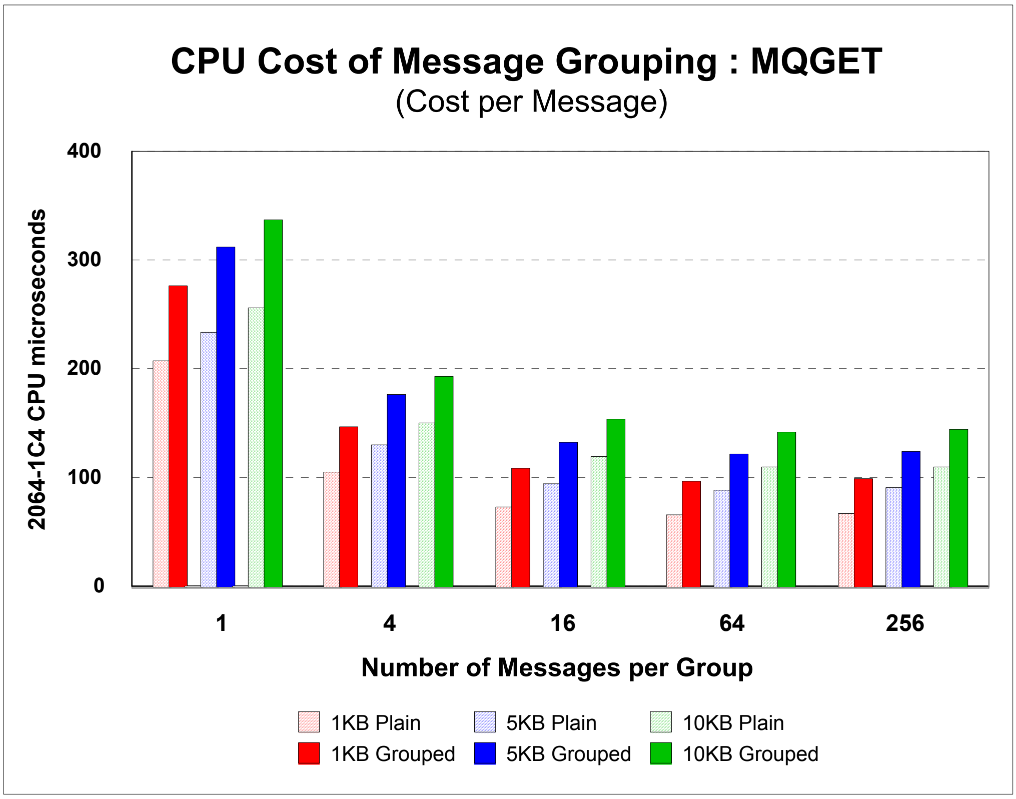


Chart: CPU Cost per Message for MQGET with Grouping - Non-persistent Messages, Shared Queue

7.0 JMS performance

JMS is now included as a separately installable feature with the V5.3 product.

A rule of thumb for increased cost per message in a z/OS workload environment is thought to be about 3 to 4 times that for the base API.

This has not been verified, but is based on the following very simple function measurements of JMS batch programs using JDK 1.3.1 compared to batch programs using the MQI and written in C.

CPU cost per 1000 byte message completely processed (put and got)

- 5 times that for the base API for non persistent messages using out of syncpoint put plus get.
- 2.5 times that for the base API for persistent messages using put/commit plus get/commit

This is significantly improved compared to the mid 2001 MA88 SupportPac supplied JMS using JDK 1.3.0 on V5.2.

These comparisons were all for messages contained within a buffer pool (no I/O to the page set) and none of the gets had to wait for a message. Either of these factors would reduce these multipliers.

8.0 Queue manager initiated expiry processing

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

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

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

For shared local queues each defined queue must be processed. A single queue manager, of those with non zero EXPRYINT in the queue sharing group, will take responsibility for this processing. If that queue manager fails or is stopped or has its EXPRYINT set to zero then another queue manager with non zero EXPRYINT will takeover. The CPU cost at each EXPRYINT interval is of order 1 CPUmillisec (9672-X27) for each shared queue defined plus any time to browse each queue and delete any expired messages. The time to browse a queue and delete any expired messages will be significantly less than using your own equivalent application because this system process avoids the overheads involved in repeated calls across the application/system boundary.

9.0 Receive Time-out

The receive time-out facility offers the ability to specify how long a channel should wait in receive mode before timing out. Receive time-out parameters are specified within the channel initiator parameter module and apply to all TCP/IP channels except SVRCONN within the queue manager. If receive time-out is enabled, the time-out interval can be specified either as an absolute value or as a function of the individual channel heartbeat interval. In the latter case, channels with heartbeating disabled will also have receive time-out disabled

9.1 Performance Implications of Receive Time-out

Receive time-out imposes a small additional CPU cost every time a channel in receive mode finds it has to wait. Typically this occurs under the following circumstances:

- *A receiver-type channel is ready to receive a message, but no messages are available*
- *A sender-type message has sent an end-of-batch indicator and waits for an acknowledgement*

The additional CPU cost occurs as a result of the channel entering the wait state. It is not dependent upon the length of time that the wait state lasts or the value of the receive time-out parameter itself.

On a 2064-1C4 processor the additional CPU cost of a single wait has been measured to be approximately 0.08 - 0.09 CPU milliseconds. The overall effect of enabling receive time-out will depend upon the channel activity profile.

The greatest impact of receive time-out is on channels with an achieved batchsize of 1. An achieved batchsize of 1 or close to 1 is quite common. Channels with low message rates, less than 30 messages of 1KB per second for instance, are likely to have an achieved batchsize of close to 1. A sender-type channel with achieved batchsize of 1 sends an end-of-batch indicator with each message and then waits for an acknowledgement before sending the next message. A receiver-type channel with achieved batchsize of 1 receives a message, acknowledges the end of batch, and then waits until the next message is available. Thus either type of channel enters a receive-wait state, and is subjected to the standard CPU overhead, with every message.

Example: Moving 5KB non-persistent messages across a channel with BATCHSZ(1), without receive time-out, consumes approximately 0.55 CPU ms at the sending end, and the same at the receiving end (this figure includes the queue manager, the channel initiator, TCP/IP and application costs involved in putting and getting the message - a real business application would normally incur higher application costs). Enabling receive time-out in this example would impose an additional .08 CPU ms per message at each end, i.e. an overhead of approximately 15%.

In the more general case, where the achieved batch size is greater than 1, the impact is smaller. Sender-type channels are subjected to the standard CPU overhead once for each completed batch; receiver-type channels are subjected to the CPU overhead at least once every batch, and possibly more, depending on the message arrival frequency. Note that the impact in the general case is dependent upon the achieved batch size rather than the BATCHSZ setting; it follows that an achieved batch size of 1, which may occur with low message rates, will generate receive time-out overheads at both ends for every message transmitted.

Example: Moving 1KB non-persistent messages across a channel with an achieved batch size of 50, without receive time-out, consumes approximately 16.5 CPU ms at each end per batch (0.33 ms per message). Enabling receive time-out in this example would impose an additional .08 CPU ms per batch at each end, i.e. an overhead of approximately 0.5%.

End of Document