Capacity Planning and Tuning for WebSphere MQ for z/OS

December, 2002

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

Property of IBM
Take Note!

Before using this document be sure to read the general information under “Notices” on page vii.
## Contents

**Part 1. SETUP AND TUNING** ................................................................. 1

**Defining your queue manager environment parameters** ........................... 1

**Log data set definitions** ................................................................. 2

- Should your archive logs reside on tape or DASD? ............................... 2
- Should your installation use single or dual logging? ............................... 2
- How many active log data sets do you need? ........................................ 3
- How large should the active logs be? .................................................. 3
- Active log placement ........................................................................... 3
- Log data set pre-format ....................................................................... 4

**Page set definitions** ........................................................................... 5

- Page set usage ..................................................................................... 5
- Size of page sets .................................................................................. 5
- Number of page sets ............................................................................. 5
- Recovering page sets .......................................................................... 5
- How often should a page set be backed up? ........................................ 6
- Achieving specific recovery targets ..................................................... 6
- Periodic review of backup frequency .................................................. 7

**Buffer pool usage and size definitions** ............................................... 8

- Buffer pool default sizes ..................................................................... 8
- Buffer pool usage ................................................................................ 8

**Shared queue setup considerations** ................................................... 10

- How many CF structures should be defined? ....................................... 11
- What size CF structures should be defined? ....................................... 11
- Increasing the maximum number of messages within a structure .......... 13
- User initiated alter processing ............................................................ 13
- How often should CF structures be backed up? ................................... 13
- When should CF list structure duplexing be used? ............................. 15
- How does use of duplexed CF structures affect performance of MQ? 15
- Non persistent shared queue message availability ............................. 17
- Is DB2 tuning important? .................................................................... 17

**Minimizing program load caused throughput effects** .......................... 18

- Frequent use of MQCONN/MQDISC - for example WLM Stored Procedures 18
- Frequent loading of message conversion tables .................................... 18
- Frequent loading of exits - for example, channel start or restart after failure 18

**Avoiding significant swapping of batch MQ applications** .................. 19

**Defining channel initiator - CSQ6CHIP parameters** ............................ 20

- ADAPS ............................................................................................ 20
- DISPS and CURRCHL ........................................................................ 20

**Tuning channels - BATCHSZ, BATCHINT, and NPMSPED** ................. 20

- Definition of BATCHSZ, BATCHINT and NPMSPED ......................... 21
- Setting NPMSPED .............................................................................. 23
- Setting BATCHSZ and BATCHINT .................................................... 23

**Tuning your MQSeries for MVS/ESA system** ..................................... 25

**V5.2 Enhanced accounting and statistics** ............................................ 25
How much extra does each waiting MQGET cost? .................................................. 102
How much extra does code page conversion cost on an MQGET? .......................... 103
MQCONN/MQDISC ................................................................................................ 103
MQOPEN/MQCLOSE (inc MQPUT1) - non-dynamic queues .................................. 103
Dynamic queue creation ....................................................................................... 103
What is the cost of creating a trigger or event message? ...................................... 103
Combining requests ............................................................................................. 103
Trace Costs ........................................................................................................ 104
Capacity planning example ................................................................................. 105

capacity planning for a requester/reply model with remote messages .............. 106
  Measurement scenario ...................................................................................... 106
  measured results .............................................................................................. 107
  resources used processing 1000-byte messages .............................................. 113
  Resources used processing 10000-byte messages ........................................... 113

capacity planning for a CICS requester / Batch reply model with local messages 114
The test scenario ................................................................................................. 114
Summary of results ............................................................................................. 114
  Why your CPU figures might differ from these .............................................. 115
Resources used when running the test scenario ................................................. 116
  1000-byte persistent messages ................................................................. 117
  1000-byte nonpersistent messages (with six server programs per server queue) 117
  1000-byte nonpersistent messages (with one server program per server queue) 119
  10 000-byte persistent messages ............................................................. 120
  10 000-byte nonpersistent messages ......................................................... 121
  100 000-byte persistent messages ............................................................ 122
  100 000-byte nonpersistent messages ....................................................... 123
  adjusting for uncollected CPU time ............................................................. 124

What happens as the transaction rate increases .................................................. 125
Log data set characteristics .................................................................................. 125

capacity planning for a CICS requester / Batch reply model with remote messages 125
Measurement scenario ....................................................................................... 126
  Number of adapters and dispatchers ............................................................ 128
  TCP/IP tuning parameters .............................................................................. 128
  Summary of measurements ............................................................................ 130
  Effect of batch size on throughput ............................................................... 131
  reduced MQGET costs when messages available immediately ....................... 131
Breakdown of costs .............................................................................................. 132
  Resources used processing 1 000-byte messages .......................................... 132
  Resources used processing 10 000-byte messages ......................................... 133
  Resources used processing 100 000-byte messages ........................................ 134

Performance implications of very large messages ............................................ 135
  General performance implications of very large messages ........................... 135
  How do CPU costs scale with increasing message size? .............................. 136
  How do elapsed times scale with increasing message size? ......................... 137
  What effect do large messages have on other MQ activity? .......................... 137
  How does a buffer pool that is too small affect large messages? ................... 137
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the maximum message rate for 100,000-byte messages?</td>
<td>184</td>
</tr>
<tr>
<td>Illustration of logging</td>
<td>180</td>
</tr>
<tr>
<td>When does a write to the log data set occur?</td>
<td>180</td>
</tr>
<tr>
<td>How data is written to the active log data sets</td>
<td>180</td>
</tr>
<tr>
<td>Single logging</td>
<td>180</td>
</tr>
<tr>
<td>Dual logging</td>
<td>180</td>
</tr>
<tr>
<td>Interpretation of key log manager statistics</td>
<td>181</td>
</tr>
<tr>
<td>Detailed example of when data is written to log data sets</td>
<td>181</td>
</tr>
<tr>
<td>Interpretation of total time for requests</td>
<td>184</td>
</tr>
<tr>
<td>What is the maximum message rate for 100,000-byte messages?</td>
<td>184</td>
</tr>
</tbody>
</table>
Notices

This document is about capacity planning and tuning for WebSphere MQ for z/OS Version 5.3, MQSeries for OS/390 Version 5.2, MQSeries for OS/390 Version 2.1 and MQSeries for MVS/ESA Version 1.2. Unless otherwise stated or implied these versions are similar for capacity planning and tuning purposes for the same function.

Note that shared queues, introduced in MQSeries for OS/390 Version 5.2, have significantly different capacity planning and tuning characteristics. If you are considering the use of shared queues then you are advised to also read both SupportPac MP1D “WebSphere MQ for z/OS Version 5 Release 3 Performance Report” and SupportPac MP1C “MQSeries for OS/390 Version 5 Release 2 Performance Report” available at http://www.ibm.com/software/ts/mqseries/txppacs/.

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 “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 of the IBM Corporation in the United States or other countries or both:

- IBM
- OS/390
- z/OS
- MQSeries
- WebSphere
- CICS
- TPNS
- DB2

The following terms are trademarks of other companies:

- Windows NT Microsoft Corporation

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>February 2001</td>
<td>Updates using MQseries for OS/390 Version 5.2 on a 9672-X37 system with ESS 2105-E20 DASD and OSA Express communications adapter. These include significant changes to the following sections</td>
</tr>
<tr>
<td>November 2001</td>
<td>Setup and Tuning Part</td>
</tr>
<tr>
<td></td>
<td>Additions</td>
</tr>
<tr>
<td></td>
<td>• Some operating system tuning opportunities</td>
</tr>
<tr>
<td></td>
<td>• Minimizing program load caused throughput effects</td>
</tr>
<tr>
<td></td>
<td>• Avoiding significant swapping of batch MQ applications</td>
</tr>
<tr>
<td></td>
<td>Updates to</td>
</tr>
<tr>
<td></td>
<td>• Use of system initiated alter processing</td>
</tr>
<tr>
<td></td>
<td>• Is DB2 tuning important</td>
</tr>
<tr>
<td></td>
<td>Addition of two new chapters in the Capacity Planning Part</td>
</tr>
<tr>
<td></td>
<td>• Capacity planning for a requester/reply model with remote messages</td>
</tr>
<tr>
<td></td>
<td>• Costs of Moving Messages Between MVS Images</td>
</tr>
<tr>
<td></td>
<td>Updates to maximum logging rates</td>
</tr>
<tr>
<td>December 2002</td>
<td>Significant changes and additions are marked with revision bars.</td>
</tr>
<tr>
<td></td>
<td>Setup and Tuning Part</td>
</tr>
<tr>
<td></td>
<td>Updates to Shared queue setup considerations for</td>
</tr>
<tr>
<td></td>
<td>• Persistent shared queue messages</td>
</tr>
<tr>
<td></td>
<td>• CF level 12 structure sizing and duplexed CF list structures</td>
</tr>
<tr>
<td></td>
<td>Addition of</td>
</tr>
<tr>
<td></td>
<td>• CSQ6CHIP definition advice</td>
</tr>
<tr>
<td></td>
<td>• Use of MQ Utilities - if possible avoid setting MAXSMSGS high</td>
</tr>
</tbody>
</table>
• Indexed queue considerations
• Queue manager initiated expiry processing

Capacity Planning Part
Updates to
• Upper Bound on persistent message capacity - DASD log data rate
• Maximum persistent message throughput using multiple processes
Bibliography

Some relevant product publications are

The MQSeries cross-platform publications:

MQSeries MQSC Command Reference, SC33-1369

MQSeries for OS/390, V5.2

MQSeries for OS/390 Concepts and Planning Guide, GC34-5650

MQSeries for OS/390 System Setup Guide, SC34-5651

MQSeries for OS/390 System Administration Guide, SC34-5652
PART 1. SETUP AND TUNING
This part has sections on

- Defining your queue manager environment parameters
  - Setting up your log data sets
  - Setting up your page sets
  - Setting up your buffer pools
  - Setting up the coupling facility for shared queues
- Some operating system tuning opportunities
  - Minimizing program load caused throughput effects
  - Avoiding significant swapping of batch MQ applications
- Walking through a simple tuning example

Defining your queue manager environment parameters

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

The following configuration options should be considered:

- Log data set definitions
- Page set definitions
- Buffer pool definitions
- Coupling Facility (CF) structure definitions, if you are using shared queues, first available with MQSeries for OS/390 Version 5.2

This chapter describes the factors that should be taken into account when designing your queue manager environment. See “Bibliography” on page 11 for definitive product publications.
Log data set definitions

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

Should your archive logs reside on tape or DASD?

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

1. Review your operating procedures before making decisions about tape or disk. For example, if you choose to archive to tape, operators must be available to mount the appropriate tapes when they are required.

2. During recovery, archive logs on tape are available as soon as the tape is mounted. If DASD archives have been used, and the data sets migrated to tape using hierarchical storage manager (HSM), there will be a delay while HSM recalls each data set to disk. You can recall the data sets before the archive log is used. However, it is not always possible to predict the order in which they will be required.

3. When using archive logs on DASD, if many logs are required (which might be the case when recovering a page set after restoring from a backup) you might require a significant quantity of DASD in order to hold all the archive logs.

4. In a low usage system or test system, it might be more convenient to have archive logs on DASD in order to eliminate the need for tape mounts.

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

1. keep as much recovery log as possible in the active logs on DASD, preferably at least enough for one day.

2. archive straight to tape

3. avoid read access of archive tapes by increasing the image copy frequency.

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

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

Should your installation use single or dual logging?

With V5.3 or V5.2 with PTF 54967 there is little performance difference between single and dual logging to write cached DASD unless the total I/O load on your DASD subsystem becomes excessive.

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

If you use devices with in-built data redundancy (for example, Redundant Array of Independent Disks (RAID) devices) you might consider using single active logging. If you use persistent messages, single logging can increase maximum capacity by 6 - 10% and can also improve response times.

If you use dual archive logs on tape, it is typical for one copy to be held locally, and the other copy to be held off-site for use in disaster recovery.
How many active log data sets do you need?

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

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

How large should the active logs be?

Your logs should be large enough so that it takes at least 30 minutes to fill a single log during the expected peak persistent message load. If you are archiving to tape, you are advised to make the logs large enough to fill one tape cartridge, or a number of tape cartridges. (For example, a log size of 1000 cylinders on 3390 DASD will fit onto a 3490E non-compacted tape with space to spare.) When archiving to tape, a copy of the BSDS is also written to the tape. When archiving to DASD, a separate data set is created for the BSDS copy. Do not use hardware compression on the tape drive as this can cause a significant impact when reading the tape backwards during recovery.

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

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

Active log placement

High persistent message throughput typically requires that the MQSeries logs are placed on fast DASD with minimum contention from other data set usage.

This used to mean there should be no other data set with significant use on the same pack as an active log. With modern RAID DASD the 3390 pack is logical with the physical data spread across multiple disk devices. However, the OS/390 UCB for the logical pack may still be a bottleneck. If UCB aliasing is available then it is possible to have up to 4 UCBs per logical pack. This is so with, for instance, ESS 2105-E20 DASD and OS/390 2.9. You can then have up to 4 busy data sets on such a logical pack with good performance for each. This can be exploited to ease any active log placement problems. For instance, you could have the current active log on the same logical pack as the preceding active log. This used to be inappropriate as the preceding log would be read for archive offload purposes while the current active log is being filled. This would have caused contention on a single UCB even to a logical pack.

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

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

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

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

Page set usage

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

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

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

Size of page sets

You should allow enough space in your page sets for the expected peak message capacity. You should also specify a secondary extent to allow for any unexpected peak capacity, such as when a build up of messages develops because a queue server program is not running. Note the application that caused page set expansion will be delayed until the expansion has completed. This can be many seconds depending on the secondary extent size.

Number of page sets

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

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

The time to recover a page set depends on

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

Recovering page sets

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

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

You can use the DISPLAY THREAD command to display the RBA of units of work and to help resolve the old ones. For information about the DISPLAY THREAD command, see the MQSeries Command Reference manual.
How often should a page set be backed up?

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

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

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

1. The time taken to react to the problem.
2. The time taken to restore the page set backup copy.

   (For example, we can restore approximately 400 cylinders of 3390 data per minute from and to RAMAC Virtual Array 2 Turbo 82 (RVA2-T82) DASD using DFDSS.)

   If you use SnapShot the time taken to perform this task is of the order of a few seconds. For further information on SnapShot, see the DFSMSdss Storage Administration Guide.

3. The time the queue manager requires to restart, including the additional time needed to recover the page set.

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

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

- The rate at which data is written to the active logs during normal processing:
  - The amount of data required on the log for a persistent message is approximately 1.3 KB more than the user message length.
  - Approximately 2.5 KB of data is required on the log for each batch of non fast messages sent on a channel.
  - Approximately 1.4 KB of data is required on the log for each batch of non fast messages received on a channel.
  - Nonpersistent messages require no log data. NPMSPEED(FAST) channels require no log data for batches consisting entirely of nonpersistent messages. The rate at which data is written to the log depends on how messages arrive in your system, in addition to the message rate. Non Fast messages received or sent over a channel result in more data logging than messages generated and retrieved locally.

- The rate at which data can be read from the archive and active logs.

   When reading the logs, the achievable data rate depends on the devices used and the overall load on your particular DASD subsystem. (For example, data rates of approximately 2.7 MB per second have been observed using active and archive logs on RVA2-T82 DASD.)

   With most tape units, it is possible to achieve higher data rates for archived logs with a large block size.

Achieving specific recovery targets

If you have specific recovery targets to achieve, for example, completion of the queue manager recovery and restart processing in addition to the normal startup time within xx seconds, you can use the following calculation to estimate your backup frequency (in hours):
Formula (A)  
\[
\frac{\text{Required restart time} \times \text{System recovery log read rate}}{\text{Application log write rate}}
\]

\[
\frac{\text{Backup frequency}}{\text{(in hours)}} = \frac{\text{(in secs)}}{\text{(in MB/sec)}} \frac{\text{(in MB/hour)}}{\text{(Application log write rate}}}
\]

For example, consider a system in which MQSeries clients generate an overall load of 100 persistent messages per second. In this case, all messages are generated locally.

If each message is of user length 1 KB, the amount of data logged per hour is of the order:

\[
100 \times (1 + 1.3 \text{ KB}) \times 3600 \text{ seconds} = \text{approximately 800 MB}
\]

where

100 = the message rate per second

(1 + 1.3 KB) = the amount of data logged for each 1 KB of persistent messages

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

\[
3600 \text{ seconds} \times 2.7 \text{ MB per second} / 800 \text{ MB per hour} = 12.15 \text{ hours}
\]

This assumes that all required log data is on RVA2-T82 DASD. (If you can only read from the log at 0.5MB per second the calculation would be every:

\[
3600 \text{ seconds} \times 0.5 \text{ MB per second} / 800 \text{ MB per hour} = 2.25 \text{ hours}
\]

If your MQSeries application day lasts approximately 12 hours, one backup each day is appropriate. However, if the application day lasts 24 hours, two backups each day is more appropriate.

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

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

\[
50 \times (2 \times (1 + 1.3 \text{ KB}) + 1.4 \text{ KB} + 2.5 \text{ KB}) \times 3600 = \text{approximately 1500 MB}
\]

where

50 = the message pair rate per second

(2 \times (1 + 1.3 \text{ KB})) = the amount of data logged for each message pair

1.4 KB = the overhead for each batch of messages received by each channel

2.5 KB = the overhead for each batch of messages sent by each channel

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

\[
1800 \text{ seconds} \times 2.7 \text{ MB per second} / 1500 \text{ MB per hour} = 3.24 \text{ hours}
\]

This assumes that all required log data is on RVA2-T82 DASD.

Periodic review of backup frequency

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

You should perform this check periodically and amend your page set backup frequency if necessary.
Buffer pool usage and size definitions

A buffer pool is an area of virtual storage in the private region of the queue manager address space. A BUFFPOOL definition gives the size in 4KB pages. Buffer pools are used to minimise I/O to and from page sets on disk. Thus both buffer pool sizes and actual usage can significantly affect the performance of queue manager operation, recovery, and restart.

Each message queue is defined to a particular storage class (STGCLASS). Each STGCLASS is assigned to a page set (PSID). Each of the up to 100 page sets (PSID(0) to PSID(99)) is assigned to a particular buffer pool (BUFFPOOL). Thus any particular queue uses one and only one bufferpool and is ultimately stored in one and only one page set.

An outline of how buffer pools are managed is given in “Introduction to the buffer manager and data manager” on page 38.

Buffer pool default sizes

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

<table>
<thead>
<tr>
<th>Definition setting</th>
<th>Test system</th>
<th>Production system</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFPOOL 0</td>
<td>1050 buffers (these were the supplied sample values until release V5.2. They are usually too small for production)</td>
<td>50 000 buffers (these are the supplied sample values from release V5.2)</td>
</tr>
<tr>
<td>BUFFPOOL 1</td>
<td>1050 buffers (old default)</td>
<td>20 000 buffers</td>
</tr>
<tr>
<td>BUFFPOOL 2</td>
<td>1050 buffers (old default)</td>
<td>50 000 buffers</td>
</tr>
<tr>
<td>BUFFPOOL 3</td>
<td>1050 buffers (old default)</td>
<td>20 000 buffers</td>
</tr>
</tbody>
</table>

Buffer pool usage

From V5.3 there are 16 buffer pools, previously there were only 4 buffer pools. We recommend you use only 4 buffer pool 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 performance of the other queues.

Buffer pools larger than 100000 pages are not usually recommended as management costs increase with increasing size.

NOTE: The optimum value for these parameters is dependent on the characteristics of the individual system. The values given are only intended as a guideline and might not be appropriate for your system. To make good use of the size recommendations (the defaults since V5.2) you should consider separating buffer pool usage as follows:

a. A buffer pool for page set zero and the page set(s) containing system related messages. Page set zero contains MQSeries objects some of which must be frequently updated. For example, queue objects have to maintain a CURDEPTH value. Ideally, keep page set zero for these system defined objects only. A crude estimate for the number of buffer pool pages
required for the system objects in page set zero is half the number of objects.
The page set containing just system related messages, say page set one, should also map to this buffer pool. System related messages are typically those in the SYSTEM.CHANNEL.SYNCO and SYSTEM.CLUSTER.* queues. **Queues that can grow large unexpectedly (for example, the dead-letter queue) are particularly inappropriate for this buffer pool.** We suggest you put such queues in the ‘everything else’ buffer pool.
This buffer pool should be large enough never to cross the less than 15% free threshold. This will avoid unnecessary reads from the page set which will effect overall MQ system performance if they are for system objects. A good starting point for the size of this buffer pool might be 50000 pages.
Alter model queue definitions to point to a storage class other than SYSTEM so that they will not map to buffer pool zero.

b. **A buffer pool for queues for your most performance critical, long lived messages.**
A good starting point for the size of this buffer pool might be 20000 pages.
Long-lived messages are those that remain in the system for longer than two checkpoints, at which time they are written out to the page set.
While it is desirable within limits to define such a buffer pool defined so that it is sufficiently large to hold all of these messages it is not recommended to exceed 50000 pages. Otherwise there may be a concentration of the necessary page set I/O at checkpoints. This might adversely affect response times throughout the system.

c. **A buffer pool for queues for short lived messages.**
A good starting point for the size of this buffer pool might be 50000 pages.
This means that you have to have only short lived messages in queues on page sets that you define to this buffer pool. Normally, the number of pages in use will be quite small, however, this buffer pool should be made large to allow for any unexpected build up of messages, such as when a channel or server application stops running.
In all cases, this buffer pool should be large enough never to cross the less than 15% free threshold.

d. **A buffer pool for everything else.**
You might not be able to avoid this buffer pool crossing the less than 15% free threshold. This is the buffer pool that you can limit the size of if required to enable the other three to be large enough. Queues such as the dead-letter queue, SYSTEM.COMMAND.* queues and SYSTEM.ADMIN.* queues should be placed here.
A good starting point for the size of this buffer pool might be 20000 pages.

See “Understanding buffer pool statistics” on page 39 for information about statistics to help monitor buffer pool usage. In particular, ensure that the lowest % free space (QPSTCBSL divided by QPSTNBUF) is never less than 15% for as many of the buffer pool usage types shown above as possible. Also ensure where possible that the buffer pools are large enough so that QPSTSOS, QPSTSTLA and QPSTNBUF remain at zero. (These are described in the MQSeries for OS/390 System Management Guide).
The effect of buffer pool size on restart time

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

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

**How many CF structures should be defined?**

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

We recommend using as few CF application structures as possible.

If any MQPUTS and MQGETS are within syncpoint a single CF application structure is recommended. Only if all the MQGETS and MQPUTS are out of syncpoint are the costs of one or multiple application structures the same.

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

For the locally driven request/reply workload using a single queue manager but with the server input queue in a different application structure to the common reply queue the unit CPU cost per request/reply increased by 21%.

**What size CF structures should be defined?**

**CSQ_ADMIN**

This CF structure does not contain messages and is not sensitive to the number or size of 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(5578) on a CF at CFCC level 12. This command also shows the current number of entries used, for instance ENTSUSED(53). We recommend you should allow about 1000 entries for each queue manager in the queue sharing group. So our example is adequate for 6 queue managers in a QSG using a CF at CFCC level 12. 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. CFCC levels before level 12 give much larger values, for example ENTSMAX(9362) on level 10.

CSQ_ADMIN usage is affected by the number of messages in each unit of work, but only for the duration of the commit for each UOW. This need only be a concern for extremely large UOWs as the minimum size structure is enough for a UOW of about 40000 messages. This is larger than the default maximum size UOW of 10000, defined by MAXUMSGS. The use of UOWs with very large numbers of messages is NOT recommended.

**Application structures**

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

- Estimate message size (including all headers)
- Round up to 256 byte boundary (subject to a minimum of 1536 bytes)
- Multiply by maximum number of messages
- Add 30% for CFCC level 12 (for other implementation considerations, this percentage can be much greater for application structures smaller than 16MB). Previous CFCC levels required the addition of 25%.
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 13.

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 700 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 244000 messages in a 0.5GB structure.

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.

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

**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.

**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 with CFCC levels prior to level 12 as there are circumstances where it is very disruptive to performance for a considerable time.**

**How often should CF structures be backed up?**
Highly available parallel sysplex systems often have stringent recovery time requirements. 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.

**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:

<table>
<thead>
<tr>
<th>1KB persistent msgs/sec to longest log</th>
<th>1KB persistent msgs/sec to 3 evenly loaded logs</th>
<th>MB/sec to longest log</th>
<th>Backup interval in minutes (= time to write 5220 MB of data to longest log)</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>1,200</td>
<td>0.97</td>
<td>90</td>
</tr>
<tr>
<td>1,200</td>
<td>3,600</td>
<td>2.9</td>
<td>30</td>
</tr>
<tr>
<td>2,200*</td>
<td>6,600</td>
<td>5.3</td>
<td>16</td>
</tr>
</tbody>
</table>

(* 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.
**When should CF list structure duplexing be used?**

CF list structure duplexing gives increased availability at a performance cost. To seriously consider duplexing you should be familiar with ‘System Managed Coupling Facility Structure Duplexing technical paper’ available at http://www.ibm.com/servers/eserver/zseries/library/techpapers/gm130103.html

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

**Availability within a given QSG may be summarised as follows**

<table>
<thead>
<tr>
<th>SIMPLEX CF Structure</th>
<th>Action on single failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSQ_ADMIN</td>
<td>Entire QSG fails.</td>
</tr>
<tr>
<td></td>
<td>The structure is rebuilt from logs at restart. All queue managers in the QSG need to restart to complete the rebuild. Only serialised applications need to wait for rebuild completion.</td>
</tr>
<tr>
<td>Application structure</td>
<td>ALL currently connected queue managers fail. On restart the structure is reallocated, all messages are lost</td>
</tr>
<tr>
<td>Application structure</td>
<td>No queue manager fails Applications using queues in that structure fail. On restart persistent messages can be recovered by any queue manager in the QSG provided that any queue manager in the QSG has done a backup and all subsequent logs are available. Alternatively the structure can be ‘recovered’ to empty.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DUPLEX CF Structure</th>
<th>Action on single failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSQ_ADMIN</td>
<td>Entire QSG stays up, z/OS recovers to duplex.</td>
</tr>
<tr>
<td>Application structure</td>
<td>ALL currently connected queue managers stay up, z/OS recovers to duplex.</td>
</tr>
<tr>
<td>Application structure</td>
<td>ALL currently connected queue managers stay up, z/OS recovers to duplex.</td>
</tr>
</tbody>
</table>

**How does use of duplexed CF structures effect performance of MQ?**

MQ operations on CF structures are typically nearly all update operations. Duplexed CF structure updates incur significant extra CF CPU and link usage. The following guidelines assume that there will be adequate total resources available. An overloaded CF is likely to cause significant performance problems for the entire sysplex.
Estimating performance for duplexed versus simplex CF structures is complex and even more than usually workload and system configuration dependent for the following reasons.

**CPU costs**

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

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

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

**Throughput**

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

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

Any throughput impacts of duplexing CF structures are because

- update operations are asynchronous for duplexed CF’s. They may be synchronous or asynchronous between z/OS and the CF for simplex CF structures, depending on an operating system heuristic decisions (first implemented in z/OS 1.2)
- the operation can only complete at the speed of the slowest link.

**CF Utilization (CF CPU)**

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

- each of the duplexed CF’s must process the operation
- plus there is synchronization between the CF’s.

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

**Duplexing the CSQ_ADMIN structure**

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

- CPU cost from 0% to 5% greater.
- Throughput decrease by 15% for 1KB non persistent messages and 5% for 10KB non persistent messages. For persistent messages throughput decrease is expected to be less than 5%.
The contribution of CSQ_ADMIN structure usage to CF utilization is usually much less than that for the application structures. Duplexing the CSQ_ADMIN structure might typically increase the MQ caused load by 10% to 20%.

The use of messages contained in more than one application structure within a unit of work increases the activity to the CSQ_ADMIN structure and so would further increase CPU and CF utilization and decrease throughput.

**Duplexing an application structure**

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

- CPU cost about 15% greater for 1KB persistent messages and 25% greater for 1KB non persistent messages.
- CPU cost about 10% greater for 10KB persistent messages and 15% greater for 10KB non persistent messages.
- Throughput decrease by 50% for 1KB non persistent messages and 35% for 10KB non persistent messages. For persistent messages throughput decrease is expected to be less than 5%. The use of messages contained in more than one application structure within a unit of work increases the activity to the CSQ_ADMIN structure and so would further decrease throughput.
- The contribution of MQ CF structure usage to CF utilization will double for the primary structures. The secondary structures will use almost as much as the primary.

**Non persistent shared queue message availability**

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

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

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

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

**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.
Minimizing program load caused throughput effects

MQSeries sometimes needs to load some programs when applications or channels start. If this happens very frequently then the I/O to the relevant program libraries can be a significant bottleneck.

Using the Library Lookaside (LLA) facility of the operating system can result in very significant improvement in throughput where program load I/O is the bottleneck.

The member CSVLLAxx in SYS1.PARMLIB specifies the LLA setup. The inclusion of a library name in the LIBRARIES statement means that a program copy will always be taken from VLF (Virtual Lookaside Facility) and hence will not usually require I/O when heavily used. Inclusion in the FREEZE statement means that there is no I/O to get the relevant DD statement concatenation directories (this can often be more I/O than the program load itself). Use the operating system FLA,REFRESH command after any changes to any of these libraries.

The following are some specific examples of when programs are loaded

Frequent use of MQCONN/MQDISC - for example WLM Stored Procedures

Every time an MQCONN is used, an MQSeries program module has to be loaded. If this is done frequently then there is a very heavy load on the STEPLIB library concatenation. In this case it is appropriate to place the SCSQAUTH library in the CSVLLAxx parmlib member LIBRARIES statement and the entire STEPLIB concatenation in the FREEZE statement.

For example:
Ten parallel batch applications running on the same queue manager were used to drive WLM (Work Load Manager) stored procedures: Each application looped 1000 times issuing 'EXEC SQL CALL Stored_Proc()'. All stored procedures ran in a single WLMSpas address space. The stored procedures issued MQCONN, MQOPEN, MQPUT (a single 1K nonpersistent message), MQCLOSE, MQDISC, but no DB2 calls were made, and were linked with the MQ/RRS stub CSQBRSTB.

1. We achieved 300 transactions a second with all of the WLMSpas's STEPLIB concatenation in LLA (in both the LIBRARIES(..) and FREEZE(..) dataset lists of the parmlib member CSVLLAxx
2. We achieved 65 transactions a second with just the LIBRARIES(..)
3. We achieved 17 transactions a second without any such tuning.

Frequent loading of message conversion tables

Each conversion type from one code page to another requires the loading of the relevant code page conversion table. This is done only once per MQCONN, however, if you have a many batch programs instances which process only a few messages each then this loading cost and elapsed time can be minimised by including the STEPLIB concatenation in both the LIBRARIES(..) And FREEZE(..) Lists.

Frequent loading of exits - for example, channel start or restart after failure

Channels, including SVRCONN thin client channels, can have four separate exits, MSGEXIT, RCVEXIT, SCYEXIT, SENDEXIT. If a significant number of channels start in a short time then a heavy IO requirement is generated to the exit library(s)

In this case the CSQXLIB concatenation must be included in the FREEZE(..) dataset lists to gain any benefit as a BLDL is done for every exit for every channel.
Avoiding significant swapping of batch MQ applications

Prior to V5.3 batch applications (those which are linkeded with CSQBSTUB or CSQRSTB or CSQRSSI) can have significant MVS swapping activity if there are MQGETs with wait or set_signal which actually do wait. Significant extra CPU cost and throughput reduction can result. To avoid this either the applications must be made non-swappable or if running on V5.2 the PTF UQ56617 should be applied. With the fix applied a swappable 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-swappable when the fix is applied would not normally be recommended.
**Defining channel initiator - CSQ6CHIP parameters**

The CSQ6CHIP parameters ADAPS and DISPS define the number of TCBS used by the channel initiator. ADAPS (adapter) TCBS are used to make MQI calls to the queue manager. DISPS (dispatcher) TCBS are used to make calls to the communications network. The CSQ6CHIP parameter CURRCHL can influence the distribution of channels over the dispatcher TCBS.

**ADAPS**

Each MQI call to the queue manager is independent of any other and can be made on any adapter TCB. Calls using persistent messages can take much longer than those for nonpersistent because of log IO. Thus a channel initiator processing a large number of persistent messages across many channels may need more than the default 8 adapter TCBS for optimum performance. This is particularly so where achieved batchsize is small because end of batch processing also requires log IO and also where thin client channels are used.

We recommend ADAPS=30 for such very heavy persistent message workloads. Using more than this is unlikely to give any significant extra benefit. We have seen no significant disadvantage in having ADPAS=30 where this is more adapter TCBS than necessary.

**DISPS and CURRCHL**

Each channel is associated with a particular dispatcher TCB at channel start and remains associated with that TCB until the channel stops. Many channels can share each TCB. CURRCHL is used to spread channels across the available dispatcher TCBS.

The first \( \min(\frac{\text{CURRCHL}}{\text{DISPS}}, 10) \) channels to start are associated with the first dispatcher TCB and so on until all dispatcher TCBS are in use. The effect of this for small numbers of channels and a large CURRCHL is that channels are NOT evenly distributed across dispatchers.

We recommend setting CURRCHL to the number of channels actually to be used where this is a small fixed number.

We recommend DISPS=20 for systems with more than 100 channels. We have seen no significant disadvantage in having DISPS=20 where this is more dispatcher TCBS than necessary.
Tuning channels - BATCHSZ, BATCHINT, and NPMSPEED

This chapter describes the channel options that can affect the throughput and cost per message for distributed queueing. To get the best from your system you need to understand the channel attributes BATCHSZ, BATCHINT and NPMSPEED introduced in MQSeries V1.2.0, and the difference between the batch size specified in the BATCHSZ attribute, and the achieved batch size. The following settings give good defaults for several scenarios:

For a synchronous request/reply model with a low message rate per channel (10s of messages per second or less), where there might be persistent messages, and a fast response is needed specify BATCHSZ(1) BATCHINT(0) NPMSPEED(FAST).

- For a synchronous request/reply model with a low message rate per channel (10's of messages per second or less), where there are only nonpersistent messages, specify BATCHSZ(50) BATCHINT(10000) NPMSPEED(FAST).
- For a synchronous request/reply model with a low message rate per channel (10's of messages per second or less), where there might be persistent messages and a short delay of up to 100 milliseconds can be tolerated specify BATCHSZ(50) BATCHINT(100) NPMSPEED(FAST).
- For bulk transfer of a pre-loaded queue specify BATCHSZ(50) BATCHINT(0) NPMSPEED(FAST).
- If you have trickle transfer for deferred processing, (the messages are typically persistent) specify BATCHSZ(50) BATCHINT(5000) NPMSPEED(FAST).
- If you are using large messages, over 100000 bytes you should use a smaller batch size such as 10, and if you are processing very large messages such as 1 MB, you should use a batch size of 1.
- For messages under 5000 bytes, if you can achieve a batch size of 4 messages per batch then the throughput can be twice, and the cost per message half that of a batch size of 1.

Definition of BATCHSZ, BATCHINT and NPMSPEED

See “MQSeries Command Reference" SC33-1369 for the full definition of these parameters.

BATCHSZ(nnn) option on channel definitions

- nnn is the MAXIMUM number of messages sent in a single batch.

BATCHINT(mmmmm) option on sending channel definitions

- mmmmm milliseconds timeout interval on batch completion
- The default is 0, which has the same effect as for releases not supporting this option.
- This interval starts from the beginning of the batch.
- Enables a trade off - reducing CPU cost, at the expense of response time for persistent messages, by increasing the achieved batch size.

NPMSPEED(FAST) option on channel definitions

- Affects ONLY the nonpersistent messages, that's why it's called NPMSPEED!
  - Transmission queues can still contain any mixture of persistent and nonpersistent messages.
- The default is NPMSPEED(FAST).
- Channels only use FAST if BOTH sender and receiver ends are defined as FAST. Some MQSeries platforms do not have NPMSPEED capability.

How batching is implemented
The text below describes the processing to send one batch of messages.

DO until BATCHSZ messages sent OR (xmitq empty AND BATCHINT expired)
  ■ Local channel gets a message from the transmission queue
    ♦ if message is nonpersistent and channel is NPMSPEED(FAST) Outside of syncpoint,
    ♦ Otherwise Within syncpoint, Adds a header to the message and sends using TCP/IP, APPC, etc.
  ■ Remote channel receives each message and puts it
    ♦ if message is nonpersistent and channel is NPMSPEED(FAST) Outside of syncpoint
    ♦ otherwise Within syncpoint
END
Perform channel synchronization logic
Thus,
• A batch will contain at most BATCHSZ messages.
• If the transmission queue is emptied before BATCHSZ is reached and BATCHINT(milliseconds) elapsed time has expired since the batch was first started then that batch will be terminated.
• The achieved batch size is the number of messages actually transmitted per batch. Typically for a single channel the achieved batch size will be small, often with just a single message in a batch, unless BATCHINT is used. If the channel is busy or the transmission queue is pre-loaded with messages, then a higher achieved batch size might obtained.
• Each nonpersistent message on an NPMSPEED(FAST) channel is available immediately it is put to a queue at the receiver, it does not have to wait until end-of-batch. Such messages are known as 'fast messages'.
• All other messages only become available at the end-of-batch syncpoint.
Note also that
• Fast messages can be lost in certain error situations, but never duplicated.
• All other message delivery remains assured once and once only.
• If the batch is terminated at BATCHSZ then an end-of-batch indicator flows with the last message.
• If the batch is terminated because the transmission queue is empty, or the BATCHINT interval expires, then a separate end-of-batch flow is generated.
• Channel synchronization logic is expensive. It includes log forces where there are persistent messages or NPMSPEED(NORMAL) channels and an end-of-batch acknowledgement flow from the receiver back to the sender. A low achieved batch size results in much higher CPU cost and lower throughput than a high achieved batch size, as these overheads are spread over fewer messages.
  For persistent messages or for all messages on channels defined NPMSPEED(NORMAL) achieving a batch size of 4 can half the cost and double the throughput on that channel compared to achieving a batch size of 1.
Setting NPMSPEED

For nonpersistent messages choosing NPMSPEED(FAST) gains efficiency, throughput and response time but messages can be lost (but never duplicated) in certain error situations. Of course nonpersistent messages are always lost in any case if a queue manager is normally stopped (or fails) and then restarted. Thus any business process using nonpersistent messages must be able to cope with the possibility of lost messages. For persistent messages NPMSPEED has no effect.

If you have applications with both persistent and nonpersistent messages which rely on message arrival sequence then you must use NPMSPEED(NORMAL). Otherwise a nonpersistent message will become available out of sequence.

NPMSPEED(FAST) is the default and is usually the appropriate choice, but don't forget that the other end of the channel must also support and choose NPMSPEED(FAST) for this choice to be effective.

Setting BATCHSZ and BATCHINT

Consider the following 3 types of application scenario when choosing BATCHSZ and BATCHINT.

1. **Synchronous Request/Reply**, where a request comes from a remote queue manager, the message is processed by a server application, and a reply sent back to the end user.
   - This usually implies a short response time requirement.
   - Response time requirements often preclude use of a non-zero BATCHINT for channels moving persistent messages.
   - Volumes on each channel might be so small that end-of-batch will occur after nearly every message even at peak loads.
   - For persistent messages, absolute best efficiency might then be achieved with BATCHSZ of 1 as there is then no separate end-of-batch flow from sender to receiver.
     - Savings of up to 20% of the CPU cost of receiving a message and sending a reply message are typical for small messages.
   - However, if your volumes and response time requirements permit then
     - Set BATCHSZ to $x$ and BATCHINT to $y$ where you typically expect $x$ or more messages in $y$ milliseconds and you can afford the up to $y$ milliseconds delay in response time on that channel.
   - **Conclusion, for channels moving any persistent messages is .. Use the defaults unless you really know better!**
   - However, nonpersistent messages on an NPMSPEED(FAST) channel are available immediately they are received regardless of BATCHINT or BATCHSZ. So a non-zero BATCHINT is appropriate for any NPMSPEED(FAST) channel carrying ONLY nonpersistent messages.
     - For example, if you expect 30 nonpersistent messages per second, set BATCHINT to 2000 (2 seconds) then you will almost always achieve a batch size of 50 (assuming BATCHSZ of 50).
     - The CPU cost saving per message moved is likely to be of order 50% versus that for achieved batch size of 1 compared to?

2. **Bulk transfer of a pre-loaded transmission queue.**
   - Usually implies high volumes, a high throughput requirement but a relaxed response time requirement (e.g. many minutes is acceptable). Thus a large BATCHSZ is desirable.
   - The default BATCHSZ of 50 will give relatively high throughput.
Higher BATCHSZs can improve throughput, particularly for persistent messages (and nonpersistent messages on NPMSPEED(NORMAL) channels).

- But might be inappropriate for very large messages sizes, where a failure in mid batch could cause significant reprocessing time.
- But do not use BATCHSZ > 100 even for messages upto 5KB.
- Do use BATCHSZ = 1 for 1MB or larger messages as anything larger tends to increase the CPU costs, and can have an impact on other applications.

**BATCHINT should be left to the default of 0.**

3. **Trickle transfer for deferred processing**

- You want to transfer the messages as they are generated as cheaply and efficiently as possible. These messages are then either stored for processing in a batch window or are processed as they arrive but it is acceptable that several seconds or minutes elapse from the time the messages were first generated.

- If possible wait until a batch size of 50 is completed. This would require that you set BATCHINT to xxxx milliseconds, where more than 50 messages are generated within xxxx milliseconds (assuming BATCHSZ greater than or equal to 50).
  - If you left BATCHINT be 0 then you would probably achieve an average batch size of less than 2 whatever the setting for BATCHSZ. In fact, it is typical that nearly all the batches would consist of just 1 message.
  - This would cost significantly more CPU and logging and some more network traffic than 1 batch of 50 messages.

- Or, consider the case where you expect an average of 20 or more messages per minute and you can accept up to 1 minute delay before these messages are processed. Then
  - If you set BATCHINT to 60000 (i.e. 1 minute) then you will achieve a batch size of 20 (on average, provided BATCHSZ greater or equal to 20)
  - If you left BATCHINT at 0 then you would probably get 20 batches of 1 message each whatever the setting for BATCHSZ.
  - 20 batches of 1 message would cost significantly more CPU and logging and some more network traffic than 1 batch of 20 messages.

- However, a very long BATCHINT is probably not appropriate as this could mean a very long unit-of-work and consequent elongated recovery time in the event of a failure. You should usually use a BATCHINT value of less than 300000 (5 minutes)

You can calculate the average batch size achieved on a given channel by using the command

```bash
DISPLAY CHSTATUS(channel) MSGS BATCHES
```

This displays both MSGS(mmm) and BATCHES(bbb). You can then calculate the achieved batch size mmm/bbb.

**Note:** Do not use `DISPLAY CHSTATUS (*)` unless really necessary when you have many channels as this is an expensive command. It might require many hundreds, or thousands of pages in buffer pool 0. Buffer pool 0 is used extensively by the queue manager itself and thus overall performance can be impacted if pages have to be written to and read from the page data sets as a result of a shortage of free buffers.
Tuning your MQSeries for MVS/ESA system

V5.2 Enhanced accounting and statistics

The enhanced accounting and statistics made available with MQSeries for OS/390 Version 5.2 offers easier performance analysis and customer chargeback. It produces extensive new accounting and statistics data including data on how each task uses each queue. The accounting data is cut at each statistics collection interval for long running tasks such as a channel, as well as at task termination. You can use this data to charge your customers for their use of MQSeries by CPU used, or by bytes processed, or to analyze and correct performance bottlenecks, and to do better capacity planning. All the data is written to SMF.

Accounting trace is enabled by the START TRACE(A) CLASS(3) command with the CPU cost typically increasing by 5% to 10% compared to that for the previously available TRACE(A) CLASS(1). TRACE(A) CLASS(1) accounting remains at typically 2% to 3% more than with no accounting. Statistics trace is enabled by START TRACE(S) with typically insignificant cost.

See SupportPac MP1B "MQSeries for OS/390 V5.2 - Interpreting accounting and statistics data". This is available at http://www.ibm.com/software/mqseries/txppacs/

Starting from a default V1.2 setup

The rest of this chapter describes the steps taken to optimize performance of an MQSeries for MVS/ESA V1.2 system that uses persistent, 1000 byte messages. Starting with the MQSeries supplied default definitions, it describes the symptoms of potential performance problems, and the actions taken to resolve them.

The supplied default definitions have been changed in MQSeries for OS/390 V5.2 to make them more appropriate for production systems. The symptoms observed and the actions taken to resolve them remain instructive for tuning purposes.

When a queue manager is lightly loaded with messages, it might appear to run well even if badly tuned. When there is a heavy load of messages, a badly tuned queue manager will exhibit poor response time and have an increased CPU cost per message. For this tuning example, measurements were taken with a constant number of messages in the system, to highlight the effects of tuning the system.

Some of the symptoms are illustrated by extracts from log manager statistics. These are standard MQSeries statistics, recorded on SMF and reported with the CSQWFSMF program (available as MQSeries SupportPac MP15).
The test scenario

The following application scenario was used to take the measurements described in this chapter.

MVS1  MVS2
TPNS  CICS  MQ  SERVER

Figure: Requester/reply application scenario

- TPNS was used to execute transactions on three CICS systems.

  The transactions are a "requester/reply" type of application, where the transaction does the following:
  1. Opens a server queue for output
  2. Opens a unique reply-to queue for input
  3. Puts a request message to the server queue specifying the name of the reply-to queue
  4. Commits the request
  5. Gets (with wait) the reply message from the reply-to queue
  6. Closes both queues
  7. Commits the work
  8. Ends

- The server application gets messages from its input queue, issues an MQPUT1 for the reply message to the specified reply-to queue, and commits the work.

Each CICS terminal put 100 messages to the server queue to preload the queue manager. 100 terminal simulations were used, leading to a total of 10000 messages in the queue manager. The response time of the transactions is not the same as the response time for a typical "put the message and wait for the reply" type application because the application gets the reply to the message put 100 transactions ago, not the message just put. However, the response times give a good indication as to how well the system performs.
Summary of results

The factors that made biggest difference to the throughput of the system during the test are shown below. These are all described in greater detail later in this chapter

- Reducing the frequency at which checkpoints occurred by increasing the size of the active log data sets, and increasing the value of LOGLOAD in CSQ6SYS.
- Increasing the size of the buffer pool to hold all of the messages; this eliminates the necessity to write pages to disk when the buffer pool fills up, and subsequently read them back when a message is required.
- Moving the log data sets so log 'n' and log 'n+1' are on different volumes to reduce I/O contention while log 'n' is being archived.

When these changes had been made, the average transaction response time decreased from 100 milliseconds to 40 milliseconds.

Comparison between the initial and final settings

The table below shows the settings that were changed during the tuning of the system.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Initial value</th>
<th>Tuned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of log data sets</td>
<td>RECORDS (1000)</td>
<td>CYL (170)</td>
</tr>
<tr>
<td>LOGLOAD parameter of CSQ6SYS</td>
<td>10000</td>
<td>500000</td>
</tr>
<tr>
<td>ALCUNIT parameter of CSQ6ARVP</td>
<td>BLK</td>
<td>CYL</td>
</tr>
<tr>
<td>PRIQTY parameter of CSQ6ARVP</td>
<td>4320</td>
<td>170</td>
</tr>
<tr>
<td>SECQTY parameter of CSQ6ARVP</td>
<td>540</td>
<td>1</td>
</tr>
<tr>
<td>Size of buffer pool 1</td>
<td>1050</td>
<td>10500</td>
</tr>
<tr>
<td>DASD volumes for 6 log data sets (3 logs with dual logging)</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Number of &quot;server&quot; queues</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Number of batch server programs serving the &quot;server&quot; queue</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>IDBACK parameter of CSQ6SYS</td>
<td>20</td>
<td>40</td>
</tr>
</tbody>
</table>

Tuning the test system

When we ran the test scenario described in "The test scenario" on page 27, we encountered several performance bottlenecks. The symptoms we saw are described below, together with the cause of the problem. We also describe the actions taken to resolve the problem, and show the resultant improvement in throughput.

Problem - Page set expansion occurs when preloading the queues

Symptom

The first time the queues were preloaded with messages, information messages about page set expansion for page set 1 were produced on the job log. The data set was expanded 8 times. When the queues were emptied and reloaded, page set expansion did not occur.

The impact of dynamic page set expansion can be seen in the following. The periods where the maximum response time is greater than two seconds is when page set expansion is occurring.
> is the maximum time within the interval
* is the mean time within the interval
< is the minimum time within the interval

<table>
<thead>
<tr>
<th>TIME</th>
<th>NUMBER OF RESPONSES</th>
<th>RESPONSE TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.13.20</td>
<td>792</td>
<td>16.13.20</td>
</tr>
<tr>
<td>16.13.30</td>
<td>1364</td>
<td>16.13.30</td>
</tr>
<tr>
<td>16.13.40</td>
<td>908</td>
<td>16.13.40</td>
</tr>
<tr>
<td>16.13.50</td>
<td>920</td>
<td>16.13.50</td>
</tr>
<tr>
<td>16.14.00</td>
<td>846</td>
<td>16.14.00</td>
</tr>
<tr>
<td>16.15.00</td>
<td>188</td>
<td>16.15.00</td>
</tr>
<tr>
<td>16.15.10</td>
<td>42</td>
<td>16.15.10</td>
</tr>
<tr>
<td>16.15.20</td>
<td>51</td>
<td>16.15.20</td>
</tr>
<tr>
<td>16.15.30</td>
<td>9</td>
<td>16.15.30</td>
</tr>
</tbody>
</table>

Figure: Response time for a CICS transaction putting messages only

This shows the effect on average response time of dynamic page set expansion. Each of the 8 expansions took about 11 seconds, and they occurred consecutively between 16:13:40 and 16:15:20.

Description of the fields:

**Time**
This is the time at the end of an interval. In this report the interval between records is 10 seconds.

**Number of responses**
This is how many transactions ended within the period.

**Response time**
The spread of the response times, with * being the mean value.

After the page set expansion had created enough space for all of the messages, the page set did not expand again when the subsequent requester/reply transactions were run.

**Action**
None, but if page set expansion continues to happen frequently, you might have to increase the size of your page set as described in the chapter about managing page sets in the *MQSeries for MVS/ESA System Management Guide (SC33-0806)*.
Problem - One server cannot keep up with the message rate

Symptom
Displaying the depth of the server queue showed that one server program could not keep up with the messages produced by the CICS transactions.

Reason
The CICS transactions were putting messages at a higher rate than the batch server could process them.

When messages are put or got from a queue there is a small degree of serialization which can impact throughput on a queue. When there are many different applications each with their own queue, one server queue may be enough with multiple servers processing this queue. When there is one very high usage application, you might get benefit from using multiple server queues and multiple server applications, although it is difficult to give a good "rule of thumb" regarding when you should do this. In real server applications there will be additional business logic that might affect throughput and the number of server programs required.

Action
1. Increase the number of servers until the server queue depth stays below 10. In this scenario, six servers were sufficient to deal with the transaction rate. Note: By default, queues cannot be shared for input, so the server program had to be stopped and the command ALTER QLOCAL(server1) SHARE DEFSOPT(SHARED) issued in order to allow multiple servers to process one queue.
2. Increase the number of server queues to four, and have six servers for each of the server queues.

Result
The response time profile of the requester/reply transaction looks like this:

<table>
<thead>
<tr>
<th>TIME</th>
<th>NUMBER OF RESPONSES</th>
</tr>
</thead>
</table>
| 16.13.30 | 897 |<--------*
| 16.13.40 | 910 |<--------*
| 16.13.50 | 893 |<--------*
| 16.14.00 | 886 |<--------*
| 16.14.10 | 921 |<--------*
| 16.14.20 | 897 |<--------*
| 16.14.30 | 929 |<--------*
| 16.14.40 | 872 |<--------*
| 16.14.50 | 903 |<--------*
| 16.15.00 | 880 |<--------*
| 16.15.10 | 889 |<--------*
| 16.15.20 | 886 |<--------*

Figure: Response time of a CICS request/reply transaction - no expansion
This is the response time profile before any tuning was done.
**Problem - Checkpoints are occurring too frequently**

**Symptom**

Several symptoms indicate that checkpoints are occurring too frequently:

- Log manager statistics show a large number of checkpoints per minute.

The following shows a printout of the log statistics over a five minute interval.

<table>
<thead>
<tr>
<th>log Write_wait</th>
<th>Write_Nowait</th>
<th>Write_Force</th>
<th>WTB</th>
<th>log Read_stor</th>
<th>Read_Active</th>
<th>Read_archive</th>
<th>WTL</th>
<th>BSDS</th>
<th>BFFL</th>
<th>BFR</th>
<th>84738</th>
<th>ALR</th>
<th>ALW</th>
<th>CIOF</th>
<th>21600</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>597753</td>
<td>3610</td>
<td>0</td>
<td>0</td>
<td>5425</td>
<td>21891</td>
<td>0</td>
<td>5425</td>
<td>21891</td>
<td>84738</td>
<td>40</td>
<td>40</td>
<td>21600</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#CHKPTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From these statistics, we can see that the number of checkpoints (#CHKPTS) in five minutes is 91. This equates to a checkpoint approximately every 5 seconds.

We recommend a checkpoint interval of about 15 minutes on a busy MQ system; with a lightly loaded system the checkpoint interval can be significantly longer. You should determine the best value by considering the effect on restart time (see “Effect of LOGLOAD on restart time” on page 179.

- Active logs fill up frequently. When a log fills up a checkpoint is taken.

The following is an extract from the console log, showing the active logs filling quickly.

16.33.15 CSQJ002I #: FULL ACTIVE LOG DATA SET
DSNAME=MQMDATA.VICA.BOX.LOGCOPY1.DS03,
STARTRBA=0000228A8000,ENDRBA=000022CDFFFF
16.33.15 CSQJ001I #: CSQJW307 CURRENT COPY 1 ACTIVE LOG DATA
SET IS DSNAME=MQMDATA.VICA.BOX.LOGCOPY1.DS01,
STARTRBA=000022CE0000,ENDRBA=000023117FFF

16.33.21 CSQJ002I #: FULL ACTIVE LOG DATA SET
DSNAME=MQMDATA.VICA.BOX.LOGCOPY1.DS01,
STARTRBA=000022CE0000,ENDRBA=000023117FFF
16.33.21 CSQJ001I #: CSQJW307 CURRENT COPY 1 ACTIVE LOG DATA
SET IS DSNAME=MQMDATA.VICA.BOX.LOGCOPY1.DS02,
STARTRBA=000023118000,ENDRBA=00002354FFFF

Figure: Extract of job log showing active logs filling frequently

**Message** CSQJ002I #: FULL ACTIVE LOG DATA SET shows that the active log data set filled up in 6 seconds and then switched to the next data set in the ring. A switch to a new log causes a checkpoint to occur.
Message CSQP018I occurs frequently, indicating that a checkpoint is beginning. The following is an extract from the console log showing the CSQP018I message occurring frequently.

16.32.51 CSQP018I @@ CSQPBCWK CHECKPOINT STARTED FOR ALL BUFFER POOLS
16.32.51 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 2, 0 PAGES WRITTEN
16.32.51 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 3, 0 PAGES WRITTEN
16.32.51 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 0, 12 PAGES WRITTEN
16.32.52 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 1, 430 PAGES WRITTEN
...
16.32.53 CSQP018I @@ CSQPBCWK CHECKPOINT STARTED FOR ALL BUFFER POOLS
16.32.53 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 2, 0 PAGES WRITTEN
16.32.53 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 3, 0 PAGES WRITTEN
16.32.53 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 0, 6 PAGES WRITTEN
16.32.54 CSQP019I @@ CSQP1DWP CHECKPOINT COMPLETED FOR BUFFER POOL 1, 326 PAGES WRITTEN

Figure: Extract of job log showing frequent checkpoints

Message CSQP018I shows a checkpoint occurring every few seconds. These checkpoints are in addition to the checkpoints caused by the active log switch, and are caused by the value of the LOGLOAD parameter of CSQ6SYSP being too small.

Action
1. CSQ6SYSP was changed to increase the LOGLOAD value from 10000 to 500000 (see "Effect of LOGLOAD on restart time on page 179").
2. The logs were increased in size to be 30 times larger than the default value, and converted from records to cylinders of 3390 (170 cylinders).
3. CSQ6ARVP was changed to match the larger log size, ALCUNIT=CYL, PRIQTY=170, SECQTY=1.
4. All of the queue manager data sets were deleted and redefined.

Result
The following shows the log manager statistics after the system was redefined to incorporate these changes.

<table>
<thead>
<tr>
<th>log Write_wait</th>
<th>0</th>
<th>Write_Nowait</th>
<th>37233</th>
<th>Write_Force</th>
<th>1073</th>
<th>WTB</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>log Read_stor</td>
<td>0</td>
<td>Read_Active</td>
<td>0</td>
<td>Read_archive</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log WTL</td>
<td>0</td>
<td>BSDS</td>
<td>32</td>
<td>BFFL</td>
<td>588</td>
<td>BFWR</td>
<td>6425</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>ALR</td>
<td>0</td>
<td>ALW</td>
<td>0</td>
<td>CIOF</td>
<td>0</td>
</tr>
<tr>
<td>LOG #CHKPTS</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Log manager statistics after LOGLOAD parameter and log size changed

From these statistics, we can see that no checkpoints were taken during the five minute interval. From the job log, we can see that the active logs filled in about 8 minutes.
Problem - Transaction response time is slow the first time logs are used

Symptom
The response time from TPNS showed a long response time, but after the log data sets had been used once, the response time was significantly improved.

<table>
<thead>
<tr>
<th>TIME</th>
<th>NUMBER OF RESPONSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.49.00</td>
<td>569</td>
</tr>
<tr>
<td>16.49.10</td>
<td>590</td>
</tr>
<tr>
<td>16.49.20</td>
<td>570</td>
</tr>
<tr>
<td>16.49.30</td>
<td>641</td>
</tr>
<tr>
<td>16.49.40</td>
<td>547</td>
</tr>
<tr>
<td>16.49.50</td>
<td>606</td>
</tr>
<tr>
<td>16.50.00</td>
<td>562</td>
</tr>
<tr>
<td>16.50.10</td>
<td>632</td>
</tr>
</tbody>
</table>

Figure: Response time before logs have wrapped for the first time

| TABLE: Response time before logs have wrapped for the first time, in milliseconds |
|---------------------------------|-----------------|----------------|-----------------|-----------------|
| Mean                           | 70              | Standard deviation | 50              | Minimum         | 10              | Maximum         | 360             |

Reason
When a log record is written in a log data set that is being used for the first time, the next record is preformatted. This effectively causes twice as much I/O, which can impact throughput. When the logs have wrapped (and so the log data sets have been used before), there is no need to preformat the records in the log data set and a higher throughput is possible.

Result

<table>
<thead>
<tr>
<th>TIME</th>
<th>NUMBER OF RESPONSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.56.00</td>
<td>962</td>
</tr>
<tr>
<td>16.56.10</td>
<td>966</td>
</tr>
<tr>
<td>16.56.20</td>
<td>939</td>
</tr>
<tr>
<td>16.56.30</td>
<td>977</td>
</tr>
<tr>
<td>16.56.40</td>
<td>952</td>
</tr>
<tr>
<td>16.56.50</td>
<td>961</td>
</tr>
<tr>
<td>16.57.00</td>
<td>954</td>
</tr>
<tr>
<td>16.57.10</td>
<td>980</td>
</tr>
</tbody>
</table>

Figure: Response time after logs have wrapped. Note that the number of responses per time interval has increased, and the response time has decreased.
Problem - The buffer pool is too small

Symptoms

1. Page set 1 was on its own DASD volume, and RMF indicated that this had a high I/O rate (40 I/O per second) for a continuous period. Because most of the messages were expected to be resident in the buffer pool, I/O to the buffer pool was expected only at checkpoint time (if at all).

2. Page set 1 was the only page set in buffer pool 1.

<table>
<thead>
<tr>
<th>BM 01</th>
<th>#BUFF</th>
<th>1050</th>
<th>#LOW</th>
<th>136</th>
<th>#NOW</th>
<th>202</th>
<th>#GETP</th>
<th>256477</th>
<th>#GETN</th>
<th>25250</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM 01</td>
<td>RIO</td>
<td>5310</td>
<td>STW</td>
<td>185186</td>
<td>TPW</td>
<td>9151</td>
<td>WIO</td>
<td>2291</td>
<td>IMW</td>
<td>4</td>
</tr>
<tr>
<td>BM 01</td>
<td>DWT</td>
<td>93</td>
<td>DMC</td>
<td>160</td>
<td>STL</td>
<td>0</td>
<td>STLA</td>
<td>7969</td>
<td>SOS</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure: Buffer pool statistics for buffer pool 1

Reason

From the information BM 01 #BUFF 1050 #LOW 136, the bufferpool statistics for buffer pool 1 show the lowest number of freebuffers was 136 out of 1050 buffers (or 13% free). When the percentage of free buffers drops below 15% of the total, pages are moved to the page sets by a task (called the Deferred Write Task) within the queue manager in order to free space in the buffer pool. The Deferred Write Task was started DWT times (93). The combination of these statistics show that the 15% threshold was reached many times, indicating that the buffer pool is too small. (The task for writing pages to disk can also be started by a checkpoint).

Action

The buffer pool was increased in size to 10500 buffers, and the queue manager restarted. The buffer pool statistics then showed #LOW was 5037 out of 10500 (or 48% free) and DWT was 0.

Result

<table>
<thead>
<tr>
<th>TIME</th>
<th>RESPONSE TIME (SECONDS)</th>
<th>NUMBER OF RESPONSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.27.10</td>
<td>&lt;=---&gt;</td>
<td>838</td>
</tr>
<tr>
<td>17.27.20</td>
<td>&lt;&lt;&lt;</td>
<td>973</td>
</tr>
<tr>
<td>17.27.20</td>
<td>&lt;&lt;&lt;</td>
<td>970</td>
</tr>
<tr>
<td>17.27.30</td>
<td>&lt;&lt;&lt;</td>
<td>920</td>
</tr>
<tr>
<td>17.27.40</td>
<td>&lt;&lt;&lt;</td>
<td>941</td>
</tr>
<tr>
<td>17.27.50</td>
<td>&lt;&lt;&lt;</td>
<td>925</td>
</tr>
<tr>
<td>17.28.00</td>
<td>&lt;&lt;&lt;</td>
<td>951</td>
</tr>
<tr>
<td>17.28.10</td>
<td>&lt;=&gt;</td>
<td>957</td>
</tr>
<tr>
<td>17.28.20</td>
<td>&lt;=&gt;</td>
<td>948</td>
</tr>
<tr>
<td>17.28.30</td>
<td>&lt;=&gt;</td>
<td>968</td>
</tr>
<tr>
<td>17.28.40</td>
<td>&lt;=&gt;</td>
<td>949</td>
</tr>
<tr>
<td>17.28.50</td>
<td>&lt;=&gt;</td>
<td>944</td>
</tr>
<tr>
<td>17.29.00</td>
<td>&lt;=&gt;</td>
<td>982</td>
</tr>
<tr>
<td>17.29.10</td>
<td>&lt;=&gt;</td>
<td>924</td>
</tr>
<tr>
<td>17.29.20</td>
<td>&lt;=&gt;</td>
<td>371</td>
</tr>
</tbody>
</table>

Figure: Response time after buffer pool size increased

Table: Response time after buffer pool size increased, in milliseconds

<table>
<thead>
<tr>
<th>Mean</th>
<th>Standard deviation</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>74</td>
<td>10</td>
<td>610</td>
</tr>
</tbody>
</table>
Problem - Log contention while archiving logs

Symptom
While a log is being archived, the transaction response time increases.

At 17:26:00 the active log filled up and switched to the next active log. The old log was archived to disk and this finished about 17:27:45.

In “Figure: Response time after buffer pool size increased” on page 34 the increased response time between 17.27.10 and 17.27.50 shows the effect of the log archive.

Reason
Because only two DASD volumes are used for the logs (one for each log copy) there was disk contention between writing to the current active log, and the reading of the previously active log to archive it.

Action
The system data sets were deleted and reallocated with the logs on separate volumes (one log per volume, three logs in the ring with dual logging, a total of six volumes).

Result

<table>
<thead>
<tr>
<th>TIME</th>
<th>NUMBER OF RESPONSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.27.50</td>
<td>957</td>
</tr>
<tr>
<td>17.28.00</td>
<td>987</td>
</tr>
<tr>
<td>17.28.10</td>
<td>947</td>
</tr>
<tr>
<td>17.28.20</td>
<td>519</td>
</tr>
<tr>
<td>17.28.30</td>
<td>878</td>
</tr>
<tr>
<td>17.28.40</td>
<td>887</td>
</tr>
<tr>
<td>17.28.50</td>
<td>879</td>
</tr>
<tr>
<td>17.29.00</td>
<td>920</td>
</tr>
<tr>
<td>17.29.10</td>
<td>909</td>
</tr>
<tr>
<td>17.29.20</td>
<td>939</td>
</tr>
<tr>
<td>17.29.30</td>
<td>885</td>
</tr>
<tr>
<td>17.29.40</td>
<td>874</td>
</tr>
<tr>
<td>17.29.50</td>
<td>944</td>
</tr>
<tr>
<td>17.30.00</td>
<td>981</td>
</tr>
</tbody>
</table>

Figure: Response time after the log data sets moved to unique volumes.

At 17:28.10 a checkpoint occurred. For a very short time (less than a few seconds) the system paused while it determined the oldest used page in the buffer pool. When this activity has completed, the backlog of MQ requests might take a few seconds to clear.

The logs wrap every 10 minutes, so the increased response time during a log archive occurs for less than 10% of the total time.
Problem - Increased transaction response time with high message rates

Symptom
At a high transaction rate, the response time sometimes increases.

Reason
When CICS connects to MQ, it attaches 8 TCBs to process MQ requests.

The flow of a CICS transaction issuing an MQ request is as follows:

1. Enters the MQ Task Related User Exit (TRUE).
2. A work request made available for dispatching.
3. If there is an idle MQ TCB, this is resumed.
4. The application then waits on an external ECB, allowing CICS to process other applications.
5. The MQ TCB dispatches the first available request and executes the MQ call.
6. When the MQ request has completed, the TCB posts the ECB and processes the next request on the work list. If there is no work to do, the TCB is suspended until there is more work for it.
7. When the application's ECB has been posted, CICS will in due time redispach the application.

If there are many concurrent requests, or the requests take a relatively long time, there could be many requests on the work list.

<table>
<thead>
<tr>
<th>TIME</th>
<th>NUMBER OF RESPONSES</th>
<th>RESPONSE TIME (SECONDS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.33.50</td>
<td>1273</td>
<td>&lt;--------*----------&gt;</td>
</tr>
<tr>
<td>8.34.00</td>
<td>2116</td>
<td>&lt;*--&gt;</td>
</tr>
<tr>
<td>8.34.10</td>
<td>2457</td>
<td>&lt;*--&gt;</td>
</tr>
<tr>
<td>8.34.20</td>
<td>2526</td>
<td>&lt;*--&gt;</td>
</tr>
<tr>
<td>8.34.30</td>
<td>2564</td>
<td>&lt;*--&gt;</td>
</tr>
<tr>
<td>8.21.20</td>
<td>1745</td>
<td>&lt;*--&gt;</td>
</tr>
<tr>
<td>8.21.30</td>
<td>1783</td>
<td>&lt;*--&gt;</td>
</tr>
</tbody>
</table>

Figure: Response time with high transaction rate

Between 08.33.41 and 08.33.57, a checkpoint occurred. For a short period transactions were suspended, and this had a knock-on effect as subsequent transactions were queued within CICS, waiting for a TCB to process the request.

The limit to the throughput is the amount of work that can run on the 8 MQ TCBs within the CICS address space.

Action
Two additional CICS regions were used.
Result

Using three CICS regions, about 300 transactions per second were processed compared to about 180 per second with only one CICS region. We can not achieve 3 * 180 transactions per second with 3 CICS regions because, as the transaction rate increases, the transaction response time increases. This is because more data is written to the log data set in each I/O, so the time for each I/O increases, and so the time for each commit increases.

Problem - Unable to start enough batch servers

Symptom

With 300 transactions a second, the depth of the server queues continued to increase, so more batch servers were started. When more than 20 servers were started, they failed with reason code 2025 (MQRC_MAX_CONNS_LIMIT_REACHED), indicating that the maximum number of connections was reached.

Action

The value of the IDBACK parameter of CSQ6SYSP was increased from the default value of 20 to 40.

Hardware and software configuration

Hardware

- Central processor
  - IBM 9627-RX5 10-way processor; this has an LSPR ratio for the default mixed workload ITR of 6.36 (relative to an IBM 9627-R15)
- Storage
  - 2000 MB central storage
  - 4096 MB expanded storage
- DASD
  - One IBM RAMAC Virtual Array 2 model T82
- Network
  - TPNS was used with three networks each of 100 terminals, each network attached to a separate CICS system

Software

- OS/390 V2.4.0
- MQSeries for MVS/ESA V1.2
- CICS/ESA V4.1
- Statistics trace enabled, with a 5 minutes interval (30 minutes is more typical for a production system)
- Accounting trace enabled
**Tuning buffer pools**

This chapter gives an outline of how buffer pools are used and what the statistics mean.

**Introduction to the buffer manager and data manager**

This describes how buffer pools are used. It will help you determine how to tune your buffer pools.

The data manager is responsible for the layout of messages within one or more 4096-byte pages, and for managing which pages are used on a page set. Multiple messages can occupy a single page, but a message always starts on a new page if it would otherwise span more than one page. A long message can span many pages. A page can contain persistent messages or non persistent messages, but not both.

The buffer manager is responsible for reading and writing these pages to the page sets, and for managing copies of these pages in memory. The buffer manager makes no distinction between persistent and non persistent messages, so both persistent and non persistent messages can be written to the page set.

A buffer pool page is written to a page set at the following times:

- At checkpoint, if it contains any change since it was last written and this is the second checkpoint to have occurred since the first such change.
- Whenever the threshold of less than 15% free buffer pool pages is reached. Pages are then written asynchronously by an internal task. This is referred to as the “15% free threshold”.
- When an application has finished with a page and there are less than 5% free pages in the buffer pool.
- At shutdown, if it contains any change since it was last written.
- From V5.3 buffer pool pages which contain non persistent messages are usually not written to a page set at checkpoint or shutdown.

A page is changed both when a message is put and when it is retrieved, because the MQGET logically deletes the message unless it is a browse.

Pages are usually written by the Deferred Write Task (DWT) asynchronously from user application activity. The DWT writes pages from the buffer pool in least recently used order (that is, from the oldest changed page).

A page is read from a page set data set into the buffer pool at the following times:

- When a message that is not already in the buffer pool is required.
- During read ahead, which is when an internal task reads a few messages into the buffer pool before an application needs them. This happens if the current MQGET does I/O to read a page and was not using MsgId or CorrelId.

Read ahead is most effective when you have a few applications getting short persistent messages with only a few messages per unit of work, because the read ahead is more likely to complete while the application waits for log I/O.

There is no direct user control on read ahead. However, you might be able to improve throughput and response time by using multiple page set data sets spread across multiple volumes to reduce I/O contention.

Differences in performance due to the size of a buffer pool depend on the amount of I/O activity between a buffer pool and the associated page set data sets. (Real storage usage, and hence paging, might also be a factor but this is too dependent on individual system size and usage to be usefully discussed here.) The unit of I/O is a page rather than a message.
The effect of message lifespan

This section discusses some message usage scenarios.

1. For messages that are used soon after they are created (that is, typically within a minute, but possibly up to 2 checkpoint intervals) and a buffer pool that is large enough to contain the high water mark number of messages, plus 15% free space:
   - Buffer pool pages containing such messages are likely to be re-used many times, meaning that relatively few pages need to be written at checkpoint and almost no pages need to be read.
   - Both CPU cost and elapsed time are minimized.

2. For messages that are stored for later batch processing:
   - All pages containing such messages are likely to be written to the page set data set because they become more than 2 checkpoints old, regardless of buffer pool size. All these pages need to be written again after the messages are used by an MQGET call, for example at the second checkpoint after the MQGET call (because the pages on the page set still contain the messages and must eventually reflect the fact that the messages are now flagged as deleted). However, if pages are reused for new messages before being written to the page set, one write operation will cover the MQGET of the old messages and the MQPUT of the new.
   - MQGET operations can still be satisfied directly from the buffer pool, provided that the pool has not reached the 15% free threshold since the required message was MQPUT.

3. In either case, if 15% free threshold is crossed, the DWT is started. This uses the least recently used algorithm for buffer pool pages to write the oldest changed buffer pool pages to the page set and make the buffer pool pages available for other messages. This means that any messages written to a page set will have to be read back from the page set if the buffer pool page is reused.
   - This is the least efficient buffer pool usage state. Elapsed time and CPU cost will be increased.
   - In many cases (for example, a single queue that is much larger than the buffer pool and is accessed in first in first out sequence) most messages will have to be read from the page set.
   - A busy buffer pool, once in this state, is likely to remain so.
Nonpersistent message processing does not require MQ log I/O and thus page set read I/Os might have greater impact on elapsed time.

A buffer pool that is large enough to keep 15% free buffers will avoid any message page reads from the page set (except after a queue manager restart). Understanding buffer pool statistics

A page in a buffer pool is in one of five states

**Unused**
This is the initial state of all pages within the buffer pool.

**Changed and in use**
The content of the page in the buffer pool is different from the matching page on the page set. Eventually the queue manager will write the pages back to the page set. The page is currently in use by an application, for example a message is being placed within it. When a large message is being put, many pages might be updated, but usually only one page will be in use at a time.

**Changed and not in use**
The page is the same as "Changed and in use" except that the page is not in use by an application.

**Uncleared and in use**
The content of the page in the buffer pool is the same as the matching page on the page set. The page is in use, for example, an application is browsing a message on the page.

**Uncleared and not in use**
The content of the page in the buffer pool is the same as the matching page on the page set, and the page is not in use by an application. If a buffer for a different page is required, the buffer page can be reassigned without its contents being written to disk.
1. The term **stealable buffers** refers to those buffers that are **unused or unchanged and not in use**. The number of stealable buffers available as a percentage of the total number of buffers affects the behavior of the buffer pool.

2. A page can only be written to disk if it is **changed and not in use**. In some circumstances, pages that are **changed and in use** are written to disk synchronously after the application has finished with the page and the page becomes **changed and not in use**. only when the I/O completes.

3. When a changed page is written to disk (so the version on disk is the same as that in the buffer pool) the page becomes **unchanged and not in use**.

The data manager issues requests for pages to the buffer manager. If the contents of a page are required, a request to get a page is issued:

- The buffer manager checks to see if the page is already in the buffer pool; if it is, the page status is set to **in use** and the address of the page is returned.

- If the page is not in the buffer pool, a stealable buffer is located, the page status is set to **in use**, the page is read in from the page set, and the address of the page is returned.

- If an update is going to be made to the page, the data manager calls the buffer manager with a SET WRITE request. The page is then flagged as **changed and in use**.

- When an application has finished with the page it releases it and, if no other application is using the page, the status is changed to **not in use**.

If the contents of the page are not required (for example, the page is about to be overwritten with a new message) a request to get a new page is issued. The processing is the same as above except, if the requested page is not in the buffer pool, a stealable buffer is located but the page is **not** read in from the pageset.

### Definition of buffer pool statistics

This section describes the buffer pool statistics. The names given are as described in the *MQSeries for OS/390 System Management Guide* (now *System Setup Guide for V5.2*). The names shown in brackets are those used by the CSQWFSMF program to print out SMF statistics. *(CSQWFSMF is available as SupportPac MP15).*

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPSTNBuf(#buff)</td>
<td>The number of pages allocated to the buffer pool in the CSQINP1 data set at MQSeries startup.</td>
</tr>
<tr>
<td>QPSTCBSL(#low)</td>
<td>The lowest number of stealable buffers during the SMF interval.</td>
</tr>
<tr>
<td>QPSTCBS(#now)</td>
<td>The number of stealable buffers at the time the SMF record was created.</td>
</tr>
<tr>
<td>QPSTGETP(getp)</td>
<td>The number of requests to get a page that were issued.</td>
</tr>
<tr>
<td>QPSTGETN(getn)</td>
<td>The number of requests to get a new page that were issued.</td>
</tr>
<tr>
<td>QPSTSTW(STW)</td>
<td>The number of SET WRITE requests that were issued.</td>
</tr>
<tr>
<td>QPSTSTRIO(RIO)</td>
<td>The number of pages that were read from the page set.</td>
</tr>
</tbody>
</table>

If the percentage of stealable buffers falls below 15% or the percentage of changed buffers is greater than 85%, the DWT is started. This task takes changed pages and writes them to the page sets, thus making the pages stealable. The task stops when there are at least 25% stealable pages available in the buffer pool.

When the status of a changed page goes from **in use** to **not in use**, and the percentage of stealable pages falls below 5% or changed pages is greater than 95%, the page is written to the page set synchronously. It becomes **unchanged and not in use**, and so the number of stealable buffers is increased.

When a checkpoint occurs, all pages that were first changed at least two checkpoints ago are written to disk, and then flagged as stealable. These pages are written to reduce restart time in the event of the queue manager terminating unexpectedly.
If a changed page was *in use* during checkpoint processing or when the DWT ran, but should have been written out, the page is written out to disk synchronously when the page changes from *in use* to *not in use*.

<table>
<thead>
<tr>
<th>QPSTDWT(DWT)</th>
<th>The number of times the DWT was started.</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPSTTPPW(TPW)</td>
<td>The total number of pages written to page sets.</td>
</tr>
<tr>
<td>QPSTWIO(WIO)</td>
<td>The number of write request.</td>
</tr>
<tr>
<td>QPSTIMW(IMW)</td>
<td>The number of synchronous write requests. (There is some internal processing that periodically causes a few pages to be written out synchronously.)</td>
</tr>
<tr>
<td>QPSTDMC(DMC)</td>
<td>The number of times pages were written synchronously to disk because the percentage of stealable buffers was less than 5% or changed pages was greater than 95%.</td>
</tr>
</tbody>
</table>

When the data manager requests a page that is not in the buffer pool, a stealable page has to be used.

<table>
<thead>
<tr>
<th>QPSTSTL(STL)</th>
<th>The number of times a page was not found in the buffer pool and a stealable page was used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPSTSOS(SOS)</td>
<td>The number of times that a stealable page was needed and there were no stealable pages available (a short on storage condition).</td>
</tr>
<tr>
<td>QPSTSTLA(STLA)</td>
<td>The number of times there was contention when getting a stealable page.</td>
</tr>
</tbody>
</table>

**Interpretation of MQ statistics**

1. If QPSTSOS, QPSTSTLA, or QPSTDMC are greater than zero you should increase the size of the buffer pool or reallocate the page sets to different buffer pools.

2. For buffer pool 0 and buffer pools that contain short lived messages:
   - QPSTDWT should be zero and so the percentage QPSTCBSL/QPSTNBUF should be greater than 15%.
   - QPSTTPW might be greater than 0 due to checkpointing activity.
   - QPSTRIO should be 0, unless messages are being read from a page set after the queue manager is restarted.
   - A value of QPSTSTL greater than 0 indicates that pages are being used that haven't been used before. This could be caused by an increased message rate, messages not being processed as fast as they were (so there is a build up of messages), or larger messages being used.
   - You should plan to have enough buffers to handle your peak message rate.

3. For buffer pools with long lived messages, where there are more messages than will fit into the buffer pool:
   - \((QPSTRIO+QPSTWIO)/\text{Statistics interval}\) is the I/O rate to page sets. If this value is high, you should consider using multiple page sets on different volumes to allow I/O to be done in parallel.
   - Over the period of time that the messages are processed (for example, if messages are written to a queue during the day and processed overnight) the number of read I/Os (QPSTRIO) should be approximately the total number of pages written (QPSTTPW). This shows that there is one disk read for every page written.
   - If the QPSTRIO is much larger than QPSTTPW, this shows that pages are being read in multiple times. This could be caused by application using `MQGET` by `MsgId` or `CorrelId`, or browsing messages on the queue using `get next`. The following actions might relieve this problem.
a. Increase the size of the buffer pool so that there are enough pages to hold the queue, in addition to any changed pages.
b. Use the INDXTYPE queue attribute (introduced in Version 1.2), which allows a queue to be indexed by MsgId or CorrelId and eliminates the need for a sequential scan of the queue.
c. Change the design of the application to eliminate the use of MQGET with MsgId or CorrelId, or the get next with browse option. Applications using long lived messages typically process the first available message and do not use MQGET with MsgId or CorrelId, and they might browse only the first available message.
d. Move page sets to a different buffer pool to reduce contention between messages from different applications.

Example of a badly tuned buffer pool

This example was taken from a production system where MQSeries Version 1.1.4 was used with the non-CICS mover. Buffer pool 0 contains only page set 0.

The system was being monitored using the ISPF interface on TSO to display information about queues and channels.

The initial symptom was that throughput to the distributed MQ systems dropped by a factor of 100.

<table>
<thead>
<tr>
<th>Field</th>
<th>Previous interval</th>
<th>Problem interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPSTNBUF</td>
<td>1050</td>
<td>1050</td>
</tr>
<tr>
<td>QPSTCBSL</td>
<td>300</td>
<td>154</td>
</tr>
<tr>
<td>QPSTCBS</td>
<td>308</td>
<td>225</td>
</tr>
<tr>
<td>QPSTGETP</td>
<td>1800000</td>
<td>23000000</td>
</tr>
<tr>
<td>QPSTGETN</td>
<td>16000</td>
<td>13000</td>
</tr>
<tr>
<td>QPSTPIO</td>
<td>0</td>
<td>310000</td>
</tr>
<tr>
<td>QPSTSTW</td>
<td>508000</td>
<td>432000</td>
</tr>
<tr>
<td>QPSTTPW</td>
<td>940</td>
<td>1938</td>
</tr>
<tr>
<td>QPSTWIO</td>
<td>59</td>
<td>107</td>
</tr>
<tr>
<td>QPSTIMWT</td>
<td>29</td>
<td>47</td>
</tr>
<tr>
<td>QPSTDWT</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>QPSTDMC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>QPSTSTL</td>
<td>84</td>
<td>732000</td>
</tr>
<tr>
<td>QPSTSTLA</td>
<td>0</td>
<td>421000</td>
</tr>
<tr>
<td>QPSTSOS</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Observations on the problem interval
1. The value for QPSTSTLA (contention when getting a stealable buffer) is 421000. This is extremely high.
2. More than half the request for a stealable buffer had contention (QPSTSTLA/QPSTSTL) = 421000/732000.
3. The number of pages read (QPSTRIO) is very high. 310000 I/O in 30 minutes is approximately 172 I/O per second (about the maximum capacity of the device).
4. QPSTDWM is zero so the buffer pool was not critically short of buffers.
5. QPSTDWT is greater than zero, QPSTCBSL/QPSTNBUF=154/1050 is 14.6%, QPSTTPW=1938, these figures are not unusual.
6. QPSTGETN is lower than the previous interval, but QPSTGETP is significantly higher. Also QPSTSTW is lower, indicating less updates. This implies that there were more requests for MQGET with browse or by Msgld or Correlld.

What was happening
1. In the mover, information on channels is held in messages on the SYSTEM.CHANNEL.SYNC.QUEUE. At the end of a batch, the messages relating to the channel are retrieved from the queue. The MQGET request uses Msgld; on Version 1.1.4, this is a sequential search of the queue, in later releases this queue can be indexed by Msgld so the sequential search is eliminated.
2. The SYSTEM.CHANNEL.SYNC.QUEUE was in page set 0 and so in buffer pool 0.
3. Normally there were sufficient stealable pages for the whole of the SYSTEM.CHANNEL.SYNC.QUEUE to be kept in the buffer pool.
4. The model queue definitions for command requests and responses pointed to page set 0.
5. For some reason (perhaps the ISPF operator asked for all information about all queues, which produced many response messages) buffer pool 0 filled up.
6. DWT processing moved the older pages out to disk and made the pages stealable.
7. When a channel reached the end of a batch, it had to read pages for the channel from the page set looking for a particular message. Because there were insufficient stealable buffers to hold the whole of the SYSTEM.CHANNEL.SYNC.QUEUE in the buffer pool, stealable pages were reused and so, for example, the buffer that held the first page of the queue was reused and was replaced with the 100th page of the queue.
8. When the next channel reached the end of a batch, it had to read the first page of SYSTEM.CHANNEL.SYNC.QUEUE from disk and re-use a stealable buffer. The stealable buffers were then "thrashing".
9. In time, the problem would gradually have corrected itself as the pages on the SYSTEM.CHANNEL.SYNC.QUEUE became changed when messages were put to and retrieved from the queue. However the ISPF panels were used to display information about the system, and pages were being written out to disk again, and the whole cycle repeated itself.

Actions taken to fix the problem
a. The SYSTEM.COMMAND.REPLY.MODEL queue was altered to use a storage class on a different page set, and so in a different buffer pool.
b. The size of buffer pool 0 was doubled. This was not strictly necessary but it allowed room for any unexpected expansion.
Use of MQ Utilities - if possible avoid setting MAXSMSGS high

The number of MQ operations within the scope of a single MQCMIT should usually be limited to a reasonably small number. For instance you should not normally exceed 100 MQPUTs within a single MQCMIT. As the number of operations within the scope of one MQCMIT increases the cost of the commit increases non linearly because of the increasing costs involved in managing very large numbers of storage blocks required to enable a possible backout.

So, for queues with many tens of thousands of messages it could be very expensive to set MAXSMSGS greater than the number of messages and use CSQUTIL utility functions like COPY or EMPTY.

APAR 54982 (PTF UQ61393 on V2.1) significantly reduces the cost of such very large units of work. This change is incorporated in V5.3.

The V5.2 Systems Administration Guide states, in the context of CSQUTIL utilities,

Syncpoints

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

MAXSMSGS should be greater than:
- The number of messages in the queue - if you are working with a single queue.
- The number of messages in the longest queue in the page set - if you are working with an entire page set.

Otherwise, the utility forcibly takes syncpoints as required and issues the warning message CSQU087I. If the function subsequently fails, the changes already committed will not be backed out. Do not simply re-run the job to correct the problem or you might get duplicate messages on your queues. Instead, use the current depth of the queue to work out, from the utility output, which messages have not been backed out. Then determine the most appropriate course of action. For example, if the function is LOAD you can empty the queue and start again or you can choose to accept duplicate messages on the queues.

The following should be added

The number of MQ operations within the scope of a single MQCMIT should usually be limited to a reasonably small number. For instance you should not normally exceed 100 MQPUTs within a single MQCMIT. As the number of operations within the scope of one MQCMIT increases the cost of the commit increases non linearly. If the CPU cost of using the CSQUTIL utilities with a very high MAXSMSGS becomes a concern and you really need to have the effect of a single commit then you should consider using an intermediate temporary queue and your own simple program with commits every say 50 messages. For example, consider the requirement to read a sequential dataset of 100,000 records and add each record as a message to an existing non-empty queue. This requires the effect of a single commit as the application cannot be just re-run in the event of failure. In this case it might be better to have simple applications which
- LOAD the records as messages to a temporary queue with a low MAXSMSGS. The temporary queue can be deleted in the event of failure and the job rerun.
- Copy this temporary queue to the target non-empty queue with say 50 MQGETs plus 50 MQPUTs per MQCMIT. This job can be rerun in the event of failure. The temporary queue is deleted on successful completion.
- In V5.2 the new command MOVE QL(tempq) TOQLOCAL(targetq) TYPE(ADD) can be used.
Indexed queue considerations

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

Indexed queues are not intended to be used as databases. Private queue indexes are created and maintained in queue manager storage, not in pagesets. These indexes must be recreated during queue manager initialization for all persistent messages in each indexed private queue. This requires that the first page of all the messages for each indexed queue be read from the pagesets. This is done sequentially by queue. For private indexed queues this will increase initialization elapsed time by the order of a few milliseconds per page read. For instance, a private indexed queue consisting of 50000 persistent messages of size 1KB increases elapsed time of initialization by about 30 seconds using Shark DASD.

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 theoretically about 13 million, which would require about 1GB of virtual storage in the queue manager address space and so is not recommended!

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

Private indexed queue rebuild at restart (from V5.3)

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 milliseconds 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.
Queue manager initiated expiry processing (from V5.3)

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.
PART 2. RESTART

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

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

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

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

There are four stages to the recovery phase of restart:

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

To understand what happens in these stages you need to understand what happens to a transaction as it updates recoverable resources (such as persistent messages) and commits them. You also need to understand what happens at when a checkpoint occurs. The next section gives a simplified description of these things. “Improved restart messages and management of long units of recovery” on page 59 describes function available from V5.2 (or with APARs) to give more messages about the restart, and to managing old units of recovery.
**What happens as a transaction processes its work**

Consider a CICS transaction that gets two persistent MQSeries messages, updates a DB2 table, and commits the work by issuing the EXEC CICS SYNCPOINT command. This will use two-phase commit because resources in both MQSeries and DB2 are updated.

<table>
<thead>
<tr>
<th>Transaction activity</th>
<th>What happens within the queue manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction starts.</td>
<td>Internal MQSeries state: Initial state</td>
</tr>
<tr>
<td>MQGET request issued.</td>
<td>1. The message is locked.</td>
</tr>
<tr>
<td></td>
<td>2. Because a unit of recovery did not exist for this transaction:</td>
</tr>
<tr>
<td></td>
<td>a. A unit of recovery is created.</td>
</tr>
<tr>
<td></td>
<td>b. The state is changed to &quot;in flight&quot;.</td>
</tr>
<tr>
<td></td>
<td>c. A &quot;Start unit of recovery&quot; (Start UR) is moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>d. The LOG RBA of this record is saved as STARTRBA in the unit of recovery record.</td>
</tr>
<tr>
<td></td>
<td>3. The &quot;message deleted&quot; flag is set in the message, and this change is moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>4. The current queue depth is decremented and this change is also moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>Final internal MQSeries state: In flight.</td>
</tr>
<tr>
<td>MQGET the second message.</td>
<td>5. The message is locked.</td>
</tr>
<tr>
<td></td>
<td>6. The &quot;message deleted&quot; flag is set in the message, and this change is moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>7. The current queue depth is decremented and this change is also moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>Final internal MQSeries state: In flight.</td>
</tr>
<tr>
<td>DB2 table update made</td>
<td></td>
</tr>
<tr>
<td>EXEC CICS SYNCPOINT issued.</td>
<td></td>
</tr>
<tr>
<td>CICS issues the first part of the two phase commit (the prepare request) to MQSeries.</td>
<td>8. A &quot;Begin Commit phase 1&quot; is moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>9. The state is changed to &quot;in commit 1&quot;.</td>
</tr>
<tr>
<td></td>
<td>10. Resource managers prepare for the commit.</td>
</tr>
<tr>
<td></td>
<td>11. The state is changed to &quot;in doubt&quot;.</td>
</tr>
<tr>
<td></td>
<td>12. An &quot;End commit phase 1&quot; record is moved to the log buffers and the RBA of this record is saved in the ENDRBA field of the unit of recovery record</td>
</tr>
<tr>
<td></td>
<td>13. The log buffers up to and including this record are forced to disk.</td>
</tr>
<tr>
<td></td>
<td>14. Returns &quot;OK&quot; to CICS.</td>
</tr>
<tr>
<td></td>
<td>Final internal MQSeries state: In doubt.</td>
</tr>
<tr>
<td>CICS issues the prepare to DB2.</td>
<td>See Note.</td>
</tr>
<tr>
<td>Providing both MQSeries and DB2 replied OK, CICS issues the second part of the two phase commit (the commit) to MQSeries.</td>
<td>1. A &quot;Phase 1 commit to Phase 2 commit&quot; record is moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>2. The state is changed to &quot;In commit 2&quot;, the transaction is now in &quot;Must complete&quot; state.</td>
</tr>
<tr>
<td></td>
<td>3. The log buffers up to and including this record are forced to disk.</td>
</tr>
<tr>
<td></td>
<td>4. The state is set to &quot;End phase 2 commit&quot;.</td>
</tr>
<tr>
<td></td>
<td>5. An &quot;End phase 2 Commit&quot; record is moved to the log buffers.</td>
</tr>
<tr>
<td></td>
<td>6. Any locked resources are unlocked.</td>
</tr>
<tr>
<td></td>
<td>7. The unit of recovery is deleted.</td>
</tr>
<tr>
<td></td>
<td>8. The state is set to &quot;Initial state&quot;.</td>
</tr>
<tr>
<td></td>
<td>9. Returns to CICS.</td>
</tr>
</tbody>
</table>

Final internal MQSeries state: Initial state.

Providing both MQSeries and DB2 replied OK, CICS issues the second part of the two phase commit (the commit) to DB2.

See Note.

**Note:** The calls to DB2 describe what logically happens. In practice, CICS optimizes the call to the last resource manager by passing the prepare and commit request together. In this example, DB2 was the last resource manager, in other cases MQSeries might be the last resource manager, and so the prepare and commit requests would be treated as one request.

If any resource manager is unable to commit, the requests are backed out.

**What happens during a checkpoint**

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

- Incomplete units of recovery.
- Recovery RBAs of all page sets.
- IMS bridge checkpoint and all Set and Test Sequence Number (STSN) information.

When a checkpoint occurs, it starts a process for writing old changed buffer pool pages to disk. These disk writes are not part of the checkpoint itself.
What happens during the recovery phase of restart

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

```
CSQJ099I @12A LOG RECORDING TO COMMENCE WITH
STARTRBA=0211B7845000
CSQR001I @12A RESTART INITIATED
CSQR003I @12A RESTART...PRIOR CHECKPOINT RBA=0211B7842C44
CSQR004I @12A RESTART...UR STATUS COUNTS
IN COMMIT=1, INDOUBT=0, INFLIGHT=1, IN BACKOUT=0
CSQR007I @12A STATUS TABLE
T CON-ID THREAD-XREF S URID DAY   TIME
- -------- ------------------------ - ------------ --- --------
S IYCPVC01 1119E2ACC3D7F1F5000032C C 0211B7843536 138 16:59:32
S IYCPVC01 1119E2ACC3D7F1F5000036C F 0211B7F88502 138 17:05:51
CSQR005I @12A RESTART...COUNTS AFTER FORWARD RECOVERY
IN COMMIT=0, INDOUBT=0
CSQR006I @12A RESTART...COUNTS AFTER BACKWARD RECOVERY
INFLIGHT=0, IN BACKOUT=0
CSQR002I @12A RESTART COMPLETED
```

Figure: Example queue manager job log showing recovery messages

Phase 1, restart begins

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

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

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

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

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

1. The last checkpoint record is located from the BSDS.

   ```
   CSQR003I @12A RESTART...PRIOR CHECKPOINT RBA=0211B7842C44
   ```

2. The recovery RBAs of each page set are read from the checkpoint records.

3. Page 0 of every page set is read to obtain the last logged RBA for the page set. The lowest RBA of all the page sets is used to determine where the log should be read from for media recovery.

4. An in-memory table is built from information in the checkpoint records of the tasks that were active at the checkpoint.

5. The log is read in a forward direction from the checkpoint to the point of failure. The in-memory table is updated as tasks complete or new tasks start.

6. Message CSQR004I displays how many tasks were in each state at the point of failure.

```
CSQR004I @12A RESTART...UR STATUS COUNTS
IN COMMIT=1, INDOUBT=0, INFLIGHT=1, IN BACKOUT=0
```
Phase 3, forward recovery

1. A list of all of the log ranges required for forward recovery is built from the list of tasks that are in doubt and in commit.
   Note: Because the log ranges between the STARTRBAs and ENDRBAs are known from the units of recovery, only the logs that contain these ranges are used. This means that some archives might not be used.

2. The range of log records for the page sets is from the earliest RBA in the page sets up to the point of failure. In normal operation the earliest RBA is within three checkpoints before the point of failure. If it has been necessary to use a backup version of a page set, the earliest RBA might be considerably earlier.

3. These ranges are combined, and a list is built of the required log ranges and the corresponding active or archive logs. The logs are read from the lowest RBA to the highest.
   Note: The logs are searched sequentially from the beginning of the data set until the start of the required log RBA is found, and then read to the end RBA.

4. For each task that is in commit, log records between the start of the unit of recovery and the ENDRBA are processed, and the changes reapplied.

5. For each task that is in doubt, log records between the start of the unit of recovery and the ENDRBA are processed, and the changes reapplied. Locks are obtained on resources as required in the same way that they are obtained during normal operation.

6. The log is read from the lowest RBA for all of the page sets and the data is replayed to rebuild the buffer pools (and the page sets if you are recovering from a backup version) as they were at the point of failure. All log records from the earliest required RBA are read, even if they do not apply to the particular page set.

7. These forward recovery steps are combined in one forward scan of the log.

8. Once this forward recovery has completed, transactions in "must-commit" state are completed. All in-doubt units of recovery stay in doubt (with any resources still locked) until they are resolved, (for example when the CICS region reconnects).

9. Message CSQR005I displays how many tasks are in commit or in doubt after forward recovery.

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

Phase 4, backward recovery.

1. The log records for in-flight or in-backout transactions are processed and any changes made by these transactions are undone.

2. Every log record is processed from the last written record, back to the earliest RBA of any transaction that was in flight or in backout. You can determine this RBA from the URID specified in message CSQR007I for records with a status of F or A.
3. Message CSQR002I is issued at completion.

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

**How long will each phase of the recovery take?**

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

1. Making the data sets available to the queue manager (for example mounting a tape or recalling a data set from HSM for archive logs).
   
   This time depends on your operational environment and can vary significantly from customer to customer.

2. Reading the active and archive logs.
   
   This depends on the hardware used, for example DASD or tape, and the speed at which data can be transferred to the processor. (On a RVA-T82 DASD we achieved between 0.6 and 2.7 MB per second depending on whether the data had been moved to tape from the disk cache.)

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

**Phase 1, restart begins**

The recovery environment is established, so this is very short.

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

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

The point of failure could be just before another checkpoint was about to be taken, in the worst case, the point of failure might be at the end of a log. You can estimate how much data is written between checkpoints by calculating the size of a log divided by the number of checkpoints in the time it takes to fill this log. (In our tests, with a log of 1000 cylinders of 3390 it took approximately 5 minutes to read to the end of this log.)
Phase 3, forward recovery

This covers three activities:

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

Recovering in-commit and in-doubt tasks

Most time is spent reading the logs between the STARTRBA and ENDRBA for in-doubt and in-commit tasks. The log is read in a forward direction until the start RBA is located, and then the records are read and processed from that point. If a unit of recovery spans more than one tape, the whole of the first tape has to be read. For in-doubt tasks, any locks are reobtained.

Applying changes to a page set if a backup copy has been used

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

To calculate the time needed to read the logs:

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

The worst case is when there is a damaged page set that was not backed up and has to be redefined. This sets the page set RBA to 0, and so all logs from the very first are required for recovery. In the example above, the previous checkpoint is 0211B7842C44. This is about 2 300 GB of data. If this can be read at 2.7 MB per second, this will take almost 10 days.

If the page set had been backed up when the queue manager was down at the RBA of 021000000000, the required range of RBAs is 0211B7842C44 - 021000000000 (about 7000 MB of data). If this can be read at 2.7 MB per second, this is about 45 minutes plus the time to read from the checkpoint to the point of failure. You also need to add the time taken to make any archive log available, and include the time to restore the page set from a backup copy.

It is significantly faster to use DFDSS dump and restore than to use IDCAMS repro. For example, for a dataset of 1600 cylinders DFDSS dump took four minutes, and IDCAMS repro took 24 minutes. The DFDSS restore took 6 minutes, and the IDCAMS repro took 24 minutes. In both cases the backup dataset was the same size as the original dataset.

Further improvements will be achieved through using a faster storage medium for the page sets and their backups. Using SHARK DASD (ESS 2105-E20), a dataset of 2000 cylinders took 2.6 minutes to dump or restore using DFDSS (a rate of approximately 9.1 MB per second), and 6 minutes using IDCAMS repro (just under 3 MB per second).

Rebuilding the buffer pools to the point of failure

To recover a buffer pool, up to three checkpoints worth of data has to be read from log. This is typically two checkpoints worth, but if the system failed when processing a checkpoint, three checkpoints worth of data needs to be processed. By having at least three active log data sets, you will ensure that these records are read from the active logs, and not archive logs.
If no page sets have been restored, rebuilding the buffer pool is usually the most significant part of forward recovery.

**Phase 4, undo any changes for tasks that were in flight or in backout (backward recovery)**

The log has to be read in a backward direction from the point of failure to the earliest URID for those tasks that were in flight or in backout (a status of F or A). Reading backwards is considerably slower than reading forwards, (by a factor of 5 on some tapes), and is even slower if data compaction is used. In the example above, if the system failed when the RBA was 0211C0000000, there is about 128 MB of data to be read. If the rate of reading backwards is 0.5 MB per second this will take about four and a half minutes.
**Example of calculation of restart time**

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

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

**Configuration**

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

When archiving to tape, a Virtual Tape System (3494-VTS B16) was used. This looks like a 3490E to OS/390. Physically, the data is written to DASD before being ultimately staged to a 3590 tape.

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

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

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

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

<table>
<thead>
<tr>
<th>Number</th>
<th>STARTRBA</th>
<th>ENDRBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>000000009FFF</td>
</tr>
<tr>
<td>2</td>
<td>00000000A000</td>
<td>000004659FFF</td>
</tr>
<tr>
<td>3</td>
<td>00000465A000</td>
<td>000008CA9FFF</td>
</tr>
<tr>
<td>4</td>
<td>000008CAA000</td>
<td>00000D2F9FFF</td>
</tr>
<tr>
<td>5</td>
<td>0000D2FA000</td>
<td>000011949FFF</td>
</tr>
<tr>
<td>6</td>
<td>00001194A000</td>
<td>000015F9FFF</td>
</tr>
<tr>
<td>7</td>
<td>000015F9A000</td>
<td>00001A5E9FFF</td>
</tr>
<tr>
<td>8</td>
<td>00001A5EA000</td>
<td>00001EC39FFF</td>
</tr>
<tr>
<td>9</td>
<td>00001EC3A000</td>
<td>000023289FFF</td>
</tr>
<tr>
<td>10</td>
<td>00002328A000</td>
<td>0000278D9FFF</td>
</tr>
</tbody>
</table>

The log data on tapes 9 and 10 is still available on the oldest two of the ring of active logs. Active logs are always used in preference to archive logs where possible.
Output from when the queue manager was restarted after it was cancelled

Phases 1 and 2, - estimate of the time needed

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

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

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

Phase 3, forward recovery - estimate of the time needed

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

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

There is one task in commit and one in doubt.

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

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

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

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

\[
0000046280C4 - 00000000A000 = 74\text{MB}
\]

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

Processing the in-commit transaction

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

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

Recovery of the buffer pools

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

Total time for this forward recovery

The time taken for forward recovery is the greater of:

- 27 seconds for the in-doubt unit of recovery plus 202 seconds for the in-commit unit of recovery
- 82 seconds for the recovery of the buffer pools

which is approaching 4 minutes, and close to the actual elapsed time.
Phase 4, backward recovery - estimate of time needed

1. The active logs were read backwards, so the archive logs on tape 10 and 9 were not needed.

<table>
<thead>
<tr>
<th>T</th>
<th>CON-ID</th>
<th>THREAD-XREF</th>
<th>S</th>
<th>URID</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:47:05</td>
<td>CSQR007I</td>
<td>08.47.05 CSQR007I @V21A STATUS TABLE</td>
<td>S</td>
<td>IYCPVC02</td>
<td>1869EE04C9D5C3D00016274C</td>
</tr>
<tr>
<td></td>
<td>CSQR007I</td>
<td></td>
<td></td>
<td>D0000046280C4</td>
<td></td>
</tr>
<tr>
<td>08:18:56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08.47.54</td>
<td>IKJ56221I DATA SET MQMDATA.TAPE.V21A1.A0000008 NOT ALLOCATED, VOLUME NO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>T</th>
<th>CON-ID</th>
<th>THREAD-XREF</th>
<th>S</th>
<th>URID</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:04:50</td>
<td>CSQR006I</td>
<td>09.04.50 CSQR006I @V21A RESTART...COUNTS AFTER BACKWARD RECOVERY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

Total restart time

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

---

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

This section describes the messages and facilities available in V5.2 or with the following APARs.

<table>
<thead>
<tr>
<th>Description</th>
<th>V120</th>
<th>V210</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages produced to identify long running units of recovery which are</td>
<td>PQ28088</td>
<td>PQ28093</td>
</tr>
<tr>
<td>no longer in the active logs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The DISPLAY THREAD command is enhanced to display the STARTRBA of all</td>
<td>PQ24346</td>
<td></td>
</tr>
<tr>
<td>threads. This will allow you to identify long running threads and resolve</td>
<td></td>
<td>In base</td>
</tr>
<tr>
<td>them.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>At system restart, if there are in-doubt or in-flight units of recovery</td>
<td>PQ25159</td>
<td>PQ27752</td>
</tr>
<tr>
<td>that start before the RBAs of the active logs, a message is produced</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>that allows the operator to commit the unit of recovery, or to let the</td>
<td>PQ28486</td>
<td></td>
</tr>
<tr>
<td>system perform forward or backward recovery as appropriate.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New message produced at startup about status of a page set and restart</td>
<td>PQ27038</td>
<td>PQ28083</td>
</tr>
<tr>
<td>progress.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Display thread command is enhanced to give the STARTRBA of the UOW**

The display thread command has been enhanced to give the STARTRBA of the UOW.

```
@!DIS THREAD(*)
06.26.14 CSQV401I @! DISPLAY THREAD REPORT FOLLOWS -
18.26.14 CSQV402I @! ACTIVE THREADS - 591
NAME       STA REQ THREAD-XREF          USERID   ASID   URID
IYCPVC01    T 4069 1869B2ACC3D7F1F50000034C PERFTASK 00D2 00000010DE78
```

This function is added by APAR PQ24346 for Version 1.2 and in is already included in Version 2.1.

**New message to show old unit of recovery no longer in active logs**

A new message CSQJ160I is produced when a STARTRBA of a unit of recovery is no longer in the active logs.

Also, if the STARTRBA of a unit of recovery is older than MAXARCH/2 archive logs, message CSQJ161I is produced. MAXARCH is the maximum number of archive logs to keep in the BSDS. It is specified by the MAXARCH keyword in CSQ6LOGP, and is displayed at startup.

If you get either of these messages you should resolve the unit of work as soon as possible. For example if you do not do this, and you had MAXARCH as 1000 and it takes 5 minutes to process each archive log, it would take more than 40 hours for the queue manager to restart after a failure.

This function is added by APARs PQ28088 for Version 1.2 and PQ28093 for Version 2.1.
**Messages give opportunity to commit old units of recovery at startup**

If the restart detects old units of recovery that are in-flight, or in-doubt, the operator is given the opportunity to commit the unit of recovery. If the operator commits all these identified units of recovery, this can significantly reduce the restart time, by reducing the scope of backward recovery of the units of work and therefore eliminating tape mounts, and processing the archive logs. If the operator does not commit the unit of recovery, backward recovery is done as usual.

This function is added by APARs PQ25159 and PQ28486 for Version 1.2 and PQ27752 for Version 2.1.

Your operational procedures must be changed to handle this new message because the queue manager waits for the reply before continuing. Your operators or automation needs to be available and be prepared to give the correct answer to this question. However you might want your administrator to make the decision rather than your operator.
**Messages which show the page set status at startup**

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Message CSQI049I Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>09.37.29</td>
<td>Page set 0 has media recovery \RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA</td>
</tr>
<tr>
<td>09.37.29</td>
<td>Page set 1 has media recovery \RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA</td>
</tr>
<tr>
<td>09.37.29</td>
<td>Page set 2 has media recovery \RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA</td>
</tr>
<tr>
<td>09.37.29</td>
<td>Page set 3 has media recovery \RBA=0000479AF9FA, checkpoint RBA=0000479AF9FA</td>
</tr>
</tbody>
</table>

This function is added by APARs PQ27038 for Version 1.2 and in PQ28083 in Version 2.1.

**Messages to show progress during forward and backward recovery**

There are 4 messages:

- **CSQR030I** This shows the maximum log range required for forward recovery.
  Note: Not every log in this range might be needed

- **CSQR031I** This message is produced approximately every 2 minutes, and shows the current log RBA being processed during forward recovery. From two of these messages, and the RBA range in message CSQR030I, you should be able to calculate the maximum time the forward recovery phase will take.
  Notes: You will also need to include the time taken to make the archive logs available.
  Active and archive logs might be on different media and thus be processed at different rates.

- **CSQR032I** This shows the maximum log range required for backward recovery.
  Note: Every log in this range will be needed.

- **CSQR033I** This message is produced approximately every 2 minutes, and shows the current log RBA being processed during backward recovery. From two of these messages, and the RBA range in message CSQR032I, you should be able to calculate the maximum time the forward recovery phase will take.
  Note: You also need to include the time taken to make the archive logs available.
This function is added by APARs PQ25159 and PQ28486 for Version 1.2 and PQ27752 for Version 2.1.

**Messages about page set recovery RBA produced at checkpoints**

A new message CSQP021I is produced during a checkpoint. It identifies the RBA stored in page 0 of the page set, and the lowest RBA of any page in the buffer pool for that page set. These values are usually the same.

This function is added by APARs PQ27038 for Version 1.2 and PQ28083 for Version 2.1.
PART 3. CAPACITY PLANNING

Capacity planning first steps

The following are some overall capacity limits which need to be considered before estimating any CPU costs.

1. There are limits to persistent message rates achievable
   - because of upper bounds to the overall data rate to an MQSeries log, see “Upper bound on persistent message capacity - DASD log data rate” on page 66. To overcome any such limit then either faster DASD or more queue managers will be required.
   - Because of upper bounds to the log I/O rate achievable by a single application instance, see “Upper bound on persistent message capacity using a single MQSeries process” on page 68.

2. There is a limit to the maximum number of messages through one queue, see “Maximum capacity using non persistent messages” on page 73.

3. There is a limit to the number of channels a queue manager can support, see “What is the maximum number of channels (or thin clients)” on page 76.

You can then either adapt specific scenario costs to your needs or use the “Some basic MQSeries API call costs (for local, non-shared queues)” on page 99 with “Costs of Moving Messages To and From MVS Images” on page 79 to build up an estimate from first principles.

Note that different scenarios may have been measured on different machines. Thus any CPU costs or capacity data need to be converted from the measured units (for example cpumillisecs on a 9672-X37) to cpumillisecs on the target machine using published LSPR ratios. An example, with caveat, is shown in “Estimating for other CPU types” on page 73.
Maximum capacity for persistent messages

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

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

This chapter has sections on

- The upper bound on the number of persistent messages that can be logged per second.
- The maximum persistent message rate through one process.
- The maximum persistent message rate through multiple processes.

What factors affect persistent message throughput?

The type of DASD used for the MQ log data sets:
- The data rate that the DASD subsystem can sustain for the MQSeries log data sets. This sets the upper limit for the MQSeries system.
- The DASD subsystem and control unit type, the amount of DASD cache, and the number of channel paths to the DASD. All will have an effect on throughput.
- Total I/O contention within the system.
- MQSeries log data set placement. Having logs on heavily used DASD volumes can reduce throughput.
- The average time to complete an I/O to the DASD volume, which depends on the amount of data to be logged as well as the DASD subsystem characteristics and MQSeries log data set placement.

Application specifics
- The rate of commits or out of syncpoint requests. This is application specific. Each commit or out of syncpoint request requires the application to wait for completion of a log IO.
- Because each application using persistent messages is likely to be I/O-bound on the log you will probably need many application instances to achieve best throughput.

Some maximum throughput examples

Using 1000 byte persistent messages and a server reply program which approximates to a 2-phase commit (by using MQGET/MQCMIT, MQPUT,MQCMIT) we have achieved

- with ESS-F20 DASD, FICON connected
  - 230 request/reply transactions (460 messages) per second through one server application driven by many local batch applications.
  - 1232 request/reply transactions (2464 messages) per second through multiple server applications driven by many local batch applications.
- with RVA2 T82 DASD
  - 55 request/reply transactions (110 messages) per second through one server application driven by thin clients.
  - 323 request/reply transactions (646 messages) per second through multiple server applications driven by thin clients.
• with ESS 2105-E20 DASD with feature 2121 (9.1 GB disks)
  ■ 111 request/reply transactions (222 messages) per second through one server application driven by thin clients.
  ■ 774 request/reply transactions (1548 messages) per second through multiple server applications driven by thin clients.

On different systems you may get different results.

Many of the results in this chapter are based on observations using MQSeries dual logs on our particular RVA2-T82 DASD subsystem - unless otherwise stated. It can sustain RMF reported I/O rates in the order of 220/second at an average response time of 2 milliseconds to both log devices. Results on other DASD subsystems or even the same type of DASD subsystem with different overall usage characteristics could be significantly different.

What is the effect of single logging on throughput?

V5.3 with logs on write cached DASD
V5.3 or V5.2 with PTF PQ54967 have similar throughput for single or dual logging on write cached DASD.

Prior releases required some writes to the dual log to be done in series with the primary log. In V5.3 (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.

Prior releases
The effect on maximum throughput is not very large for modern DASD. For instance, using MQSeries V2.1 the maximum throughput on ESS 2105-E20 DASD with feature 2121 (9.1 GB disks), we can achieve is

• Single logging  - 7.54 MB/second
• Dual logging  - 7.10 MB/second

Single logging has an increase of 6% in maximum throughput over dual logging. It is believed that older device types tend to have a somewhat larger difference.

However, the effect on more typical throughput rates can be significant. With short messages and low throughput rates most writes to the secondary log data set would be in series. Persistent message throughput can be improved by up to 30% for 1KB persistent messages by using single logging rather than dual logging.

The QJSTSERW log statistic shows the number of serial writes, see SupportPac MP1B Interpreting accounting and statistics data.
Upper bound on persistent message capacity - DASD log data rate

The maximum total MQSeries system capacity for persistent messages is bounded by the maximum data rate sustainable to the DASD subsystem where the MQSeries logs reside. For MQSeries with dual logging, the maximum sustainable data rate for messages put at one message per commit and got at one message per commit is about

<table>
<thead>
<tr>
<th>Log data set DASD type</th>
<th>1KB messages</th>
<th>5KB messages</th>
<th>1 MB messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVA-T82</td>
<td>2.3 MB/sec</td>
<td>2.8 MB/sec</td>
<td></td>
</tr>
<tr>
<td>ESS E20 with ESCON</td>
<td>5.3 MB/sec</td>
<td>7.1 MB/sec</td>
<td></td>
</tr>
<tr>
<td>ESS F20 with ESCON</td>
<td>7.1 MB/sec</td>
<td>10.2 MB/sec</td>
<td>11.3 MB/sec</td>
</tr>
<tr>
<td>ESS F20 with FICON</td>
<td>7.4 MB/sec</td>
<td>13.0 MB/sec</td>
<td>15.6 MB/sec</td>
</tr>
</tbody>
</table>

Will striped logs improve performance?

Switching to active logs which use VSAM striping can lead to improved throughput in situations where performance is being constrained by the log data rate. The benefit obtained from using VSAM striping varies according to the amount of data being written to the log on each write. For example, if the log dataset has been set up with 4 stripes, a log write carrying a small amount of data such that only one stripe is accessed will gain no benefit at all, while a log write carrying sufficient data to access all 4 stripes will gain the maximum benefit.

The increased logging rate achieved by using striped active logs will result in the log filling more quickly. However, the time taken to archive a log dataset is unchanged. This is because archive log datasets must not be striped as the BDAM backwards reads required during recovery are not supported on striped datasets. Thus the possibility of needing to reuse a log dataset before its previous archive has completed is increased. It may therefore be necessary to increase the number or size of active log datasets when striping is used. If you attempt to sustain these maximum rates to striped logs for long enough then eventually you will fill all your active logs with consequent unacceptable performance.

On our system, using ESS 2105-F20, we achieved the following logging rates for striped logs until all the active logs were full:

<table>
<thead>
<tr>
<th>Striped log data set DASD type</th>
<th>1KB messages</th>
<th>5KB messages</th>
<th>1 MB messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESS F20 with 4 stripes (ESCON)</td>
<td>7.5 MB/sec</td>
<td>12.7 MB/sec</td>
<td>19.4 MB/sec</td>
</tr>
<tr>
<td>ESS F20 with 4 stripes (FICON)</td>
<td>8.4 MB/sec</td>
<td>14.2 MB/sec</td>
<td>18.5 MB/sec</td>
</tr>
</tbody>
</table>

In summary, striped logs are most likely to be of use where there is a reasonably predictable amount of large messages in a particular time period such that the total amount of data to be logged does not cause the active logs to be filled.

Will striped logs affect the time taken to restart after a failure?

The recovery process will need to read active logs and this is significantly quicker with striped datasets, particularly for the backward recovery phase. It may also involve reading archived log datasets which cannot be striped. Thus any use of archive log datasets during recovery will not be quicker. It is possible to minimise or even eliminate the possibility of an archive log being required during recovery. This requires pageset backup at appropriate intervals and appropriate reaction to any CSQJ1601 messages concerning long running units of recovery with a STARTRBA no longer in active logs.

How can we estimate the required log data rate for a system?
The amount of data written to an MQSeries log for a persistent message which is MQPUT and committed then MQGET and committed is approximately:

\[
\text{User message length + length(all headers) + 1000 bytes}
\]

Thus, for a 1000 byte persistent message put to and got from a local queue approximately 2300 bytes of data will be written to the MQSeries log.

Using the maximum sustainable DASD data rates given above, for 1000 byte messages we estimate that up to 2.3 MB / 2300 bytes = 1000 persistent messages/second can be processed on our RVA2-T82 DASD subsystem; we have achieved this throughput in one measurement scenario with enough concurrent processes, though there was an increased response time. On other DASD subsystems you may get a different maximum.

For long messages the log data requirement is further increased by about 150 bytes per page occupied by the message and all its headers. For example a 10000 byte user message requires three 4KB pages. Approximately 10000 + header length + 1000 + (3*150) = 11750 bytes of data will be required on the MQ log for such a message on a local queue.

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

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

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

For throughput estimating purposes assume

- For MQGET the log data requirement is about 500 bytes for messages up to a user length of 10 KB. This increases linearly to about 1300 bytes for messages of user length 100 KB.
- For MQPUT the actual message, including header data, is placed on the log. To estimate MQPUT requirement calculate
  \[
  \text{Total log requirement (as above) - MQGET log requirement}
  \]

Please note that the above calculations only give throughput estimates. Log activity from other MQSeries processes can affect actual throughput.
**Upper bound on persistent message capacity using a single MQSeries process**

An MQSeries process is, for example

- a batch job
- a CICS/MQSeries adapter
- a channel initiator (CHINIT) adapter

We achieve up to 55 request/replies (110 messages of 1000 bytes) per second through a single server reply batch job on our RVA2 T82 DASD subsystem.

**What limits the capacity through a single MQSeries process?**

Within the overall MQSeries log data rate limit on a particular system there is usually a much lower limit to the throughput of any individual process. This is because each such process has to wait for I/O every time it requires to commit something to the log. The number of commits is application specific. However, the wait time for each commit will depend on the total load on the log rather than just the load from a particular process. (Of course, it will also depend on the total load on the DASD subsystem). The total load on the log is dependent on the volume of message data (size times number of messages).

The number of waits for I/O to the MQSeries log is not dependent on the number or volume of persistent messages, but on the application specific number of log commits. An application commits to the log at each

- out-of-syncpoint call
- explicit MQCMIT call
- implicit MQCMIT call as part of a CICS or IMS syncpoint
- two-phase commit 'prepare',
  - when MQSeries is used by a CICS or IMS transaction which also updates its own resources
  - when MQSeries uses RRS to coordinate work across multiple resource managers.

I/O to the MQSeries log is done in 4 KB page units, so the number of pages to be written at a commit will increase according to message data volume. This will tend to increase the time a process has to wait for a commit to the log.

We have measured that the maximum commit rate a single process can sustain, using our RVA2-T82 DASD subsystem, with a dual log which is already moderately busy with other work, is about

- 120 commits/sec for messages of length 100 user bytes.
- 110 commits/sec for messages of length 1000 user bytes.
- 90 commits/sec for messages of length 5000 user bytes.

*Be careful in any use of this information to understand differences between transactions/second, log commits/second, and messages/second.*

The upper bound on throughput for any particular single process where the dual log is only moderately busy is approximately

\[
\text{maximum commit rate} / \text{number of commits required.}
\]
For example, our thin clients driven request/reply scenarios have a total of 2 persistent messages and
4 log commits per request/reply transaction.

- The server reply program part of each transaction is coded in such a way as to require 2 log
  commits. This is to simulate the worst case of a CICS transaction doing two-phase commits.
- The thin clients, via many CHINIT adapter processes, are generating enough work to keep the
  MQSeries log moderately busy.
- The single server reply program should be able to play its part in processing about \( \frac{110}{2} = 55 \)
  transactions/sec (or 110 messages/sec). We have measured 54 transactions/sec.

However, as the log becomes increasingly busy for whatever reason, throughput for each process will
become further limited as the log commit wait time lengthens. For example:

- with multiple servers and a message rate of 640 messages of 1000 bytes per second, the capacity
  of one instance of a server was about 20 request/replies (40 messages) per second.
- where 1 MB persistent messages were being processed by other processes, the throughput of
  1000 byte messages through one server was around 4 request/replies (8 messages) per second.
**Maximum persistent message throughput using multiple processes**

### Local batch driven request/reply

The measured scenario was multiple local batch requesters driving a set of identical reply programs.

The requesters put and get out of syncpoint. Thus each requester commits (implicitly) twice per request/reply transaction. The put must be committed separately from the get so that the message can be made available to the reply program to generate the reply message. There were 90 requester instances for each measurement.

Two sets of server programs were used.

1. **The first use get and put out of syncpoint. This is to simulate the case of, for instance, a CICS transaction that updates some of its own resources and uses a full two-phase commit. Both this case and a real two-phase commit require two forced IOs to the log data sets.**

2. **The second use get and put within syncpoint. Thus only one forced log IO is required**

These results were obtained with V5.3 on a 2064-1C4 running z/OS 1.4. The log DASD was FICON connected ESS-F20

<table>
<thead>
<tr>
<th>Maximum transaction throughput with dual logging on FICON connected ESS-F20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Locally driven request/reply transactions</strong></td>
</tr>
<tr>
<td><strong>Msg size</strong></td>
</tr>
<tr>
<td>1,000</td>
</tr>
<tr>
<td><strong>Locally driven request/reply transactions</strong></td>
</tr>
<tr>
<td><strong>Msg size</strong></td>
</tr>
<tr>
<td>1,000</td>
</tr>
</tbody>
</table>

### Thin client driven request/reply

The measured scenario was multiple thin client requesters driving a batch server reply program (or set of identical batch programs).

The thin clients put and get out of syncpoint. Thus each client commits (implicitly) twice per request/reply transaction. The put must be committed separately from the get so that the message can be made available to the reply program to generate the reply message.

The server programs get and put out of syncpoint. This is to simulate the worst case of a CICS transaction that updates some of its own resources and uses a full two-phase commit, which is equivalent to two commits. However in a standard server application, with a get/put/commit, we would expect to achieve a higher throughput because of the fewer commits, but each server request would take longer because of business logic which reduces the throughput, so our server is perhaps typical of what a real application does.

When a single batch server program is used in this scenario it becomes the limiting factor. With multiple server programs the total capacity could be limited by the number of adapters defined in the channel initiator (CSQ6CHIP parameter ADAPS). In our 16 server test we ran with 30 adapters instead of the default of 8.

Note that the CICS/MQSeries interface has a fixed number (8) of CICS/MQSeries adapters. This does not limit the number of CICS/MQSeries transactions that can be run concurrently but the capacity of any individual CICS region is ultimately limited for persistent message work by this limit of 8 MQSeries processes. Many CICS regions can be employed using one MQSeries queue manager if necessary.
Measurement configuration

A simple thin client driven request/reply scenario was used. The requesters were represented by a number of Windows NT MQI thin client applications, each of which MQPUT a single message to the request queue then MQGET-waited on a reply queue; both calls outside of syncpoint.

A batch server reply program simply MQGET-waited on the request queue and returned messages to the common reply queue using MQPUT, that is, there was no business application logic. Both MQI calls were outside of syncpoint, thus there were two implicit commits in the server program per request/reply transaction.

Each transaction requires

- 2 commits for the thin client calls, each performed by one of the channel initiator adapters.
- 2 commits to the dual log by the server program.

<table>
<thead>
<tr>
<th>Table: Maximum transaction throughput with dual logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thin client driven request/reply transactions - 2 persistent messages / transaction</td>
</tr>
<tr>
<td>Message Length</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>100 (RVA2-T82)</td>
</tr>
<tr>
<td>1000 (RVA2-T82)</td>
</tr>
<tr>
<td>1000 (ESS 2105-E20)</td>
</tr>
<tr>
<td>5000 (RVA2-T82)</td>
</tr>
</tbody>
</table>

The server program was defined as non-swappable, this avoids potentially significant swap CPU cost (RCT time) see “Additional CPU used by batch servers” on page 125

Remote queue manager driven request/reply

We managed to achieve about 200 request/reply transactions (400 messages) per second with 1000 byte persistent messages, using RVA2-T82 DASD. We would expect significant improvement if ESS 2105-E20 DASD were to be used.

One or more server applications running on a 'server' queue manager were driven by multiple request applications on 10 'client' queue managers (running on different MVS images to the 'server' queue manager).

The server program used here was the same as for the thin client measurements above.

Each transaction requires

- 2 commits to the dual log by the server program.
- In this case the log I/O required by the channels for assured message delivery is important and varies according to the achieved batch size as each end of batch adds its own data and more importantly log commits.
  - An achieved batch size of 1 is typical of many request/reply workloads unless individual channels are very busy.
  - An achieved batch size of more than 3 is not often exceeded.
  - An achieved batch size of 100 would probably require a concentrator MQSeries node and a relaxed response time requirement.
We forced the achieved batch size to the desired amount by using the BATCHINT channel parameter. See "Tuning channels - BATCHSZ, BATCHINT, and NPMsPEED" on page 22 for a discussion of achieved batch size and channel tuning parameters.

<table>
<thead>
<tr>
<th>Achieved Batch Size</th>
<th>1 * Server</th>
<th>16 * Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29 (58)</td>
<td>41 (82)</td>
</tr>
<tr>
<td>3</td>
<td>37 (74)</td>
<td>84 (168)</td>
</tr>
<tr>
<td>100</td>
<td>61 (122)</td>
<td>203 (406)</td>
</tr>
</tbody>
</table>

Table: Maximum transaction throughput with dual logging (on RVA2-T82 DASD)
Maximum capacity using non persistent messages

Throughput for non persistent messages is ultimately limited by the CPU power and the use of internal locks and latches which can limit the exploitation of multiple CPU engines.

What is the maximum message rate (put & get) through a single queue?

Using a 9672-X37 system running OS/390 V2.10 and MQSeries for OS/390 V5.2 we could sustain the following non persistent message rates. Sustained means, in this case, that messages are MQPUT and MQGET at about the same rate so that the queue does not get very large. If you run one MQPUT job with one MQGET job, both flat out with no business logic, then the queue will build up because of internal locking considerations.

We run three MQPUT jobs in parallel with a single MQGET job to obtain these results. The MQPUT jobs include a short wait after so many messages so that the MQGET can keep up. Both MQPUT and MQGET calls are out of syncpoint.

<table>
<thead>
<tr>
<th>Message size</th>
<th>Message rate / sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>4,100</td>
</tr>
<tr>
<td>5,000</td>
<td>4,000</td>
</tr>
<tr>
<td>10,000</td>
<td>2,950</td>
</tr>
</tbody>
</table>

Estimating for other CPU types

We expect these particular rates to scale approximately with LSPR (Large Systems Performance Reference - SC28-1187) ITR ratios across 390 machine ranges. For instance if we take ITR ratios for the “Mixed” workload column of the “zSeries LSPR ITR ratios for IBM processors” section of LSPR we see ITR ratio 2.75 for a zSeries 900 machine type 2064-1C3 and an ITR ratio of 1.86 for a 9672-X37.

An estimate for maximum message rate for 1000 byte nonpersistent messages on such a zSeries 900 class machine is therefore about 4100 * 2.75 / 1.86 = 6061 messages/sec.

Caveat

Use of just ITR ratios is possible on the assumption that CPU power is the limiting factor. This is often true for non persistent messages. However, other limitations may become significant. For instance, inadequate BUFFERPOOLS could mean that page set IO becomes the limiting factor.

For persistent messages MQSeries log data rate is often the limiting factor.

There may be other limitations beyond the scope of MQSeries whatever the message persistence used. For instance network bandwidth when transmitting messages or using MQSeries thin clients. You also need to factor in all business logic costs and constraints as there is none in our workloads.

What is the maximum message rate (put & get) through a single shared queue?

Using the same technique as above we see

<table>
<thead>
<tr>
<th>Shared Q Message size</th>
<th>1 queue manager Message rate / sec</th>
<th>2 queue managers Message rate / sec</th>
<th>3 queue managers Message rate / sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>5,850</td>
<td>10,900</td>
<td>16,550</td>
</tr>
<tr>
<td>5,000</td>
<td>3,200</td>
<td>6,100</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>2,900</td>
<td></td>
<td>5,400</td>
</tr>
</tbody>
</table>
**Throughput for request/reply pairs of local queues**

The following message rates are for locally driven request/reply scenarios.

Each request/reply scenario uses:
- one or more request/reply applications, each of which uses
  - a pair of queues (a common server queue and a common reply queue)
  - one or more reply programs using that pair of queues
  - one or more requester programs per reply program

<table>
<thead>
<tr>
<th>Locally driven Request/Reply on 9672-X37</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q pairs</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

**Throughput for request/reply pairs of shared queues**

Using the same scenario as above

<table>
<thead>
<tr>
<th>SharedQ - single queue manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q pairs</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SharedQ - 1 queue manager on each of 2 OS/390 images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q pairs</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Where 2+2 means each queue manager ran 2 requesters to each pair of queues
**Throughput for request/reply driven by messages over channels**

We use a previously published measurement for this which we use to show expected, *but not verified*, throughput on a 9672-X37 using LSPR ITR ratios (see “Large Systems Performance Reference” SC28-1187).

**Previously published measurement result**

We have observed 850 remote queue manager driven request/reply transactions per second (1700, 1000 byte nonpersistent messages/sec) while using 85% of the total CPU power of a 9672-R55.

**Estimate for throughput on 9672-X37 using LSPR ITR ratios**

We use the “Mixed” workload column of ITR ratios from the “OS/390 V2 R4 LSPR ITR ratios for IBM processors” section of SC28-1187.

The measurement was on a 9672-R55 which has an ITR ratio of 4.23 while a 9672-X37 has an ITR ratio of 7.62.

The measurement used 85% CPU of the 5 engined 9672-R55, so the exploitation of multiple engines does not appear to be a problem in this case. We can simply extrapolate the observed 850 transaction/sec on a 9672-R55 to an estimated $850 \times \frac{7.62}{4.23} = 1531$ transactions/sec.

So, for capacity planning purposes we can reasonably estimate that a 9672-X37 could sustain about 1500 transactions/sec (3000 nonpersistent messages/sec). **Of course, there could be constraints other than CPU power to consider, for instance network bandwidth and see “Caveat “ on page 73 above.**

**Measurement configuration**

A simple remote queue manager driven requester/reply scenario was used. We had 10 requester queue managers spread over 3 MVS images, separate to the measured MVS server reply queue manager.

We used Version 2.1 cluster definitions such that the server queue was defined in a cluster with each queue manager having a single CLUSRCVR channel in that cluster. The reply queues were not part of the cluster. (We expect this to be the normal clustering definition style for queue manager to queue manager request/reply scenarios). This definition results in a single sender and receiver channel pair between each driver queue manager and the server queue manager.

2 batch MQI requester applications were used per driver queue manager. 5 MQI batch server reply applications were used in the server queue manager being measured. The requester applications looped continually doing 50 PUTs out of syncpoint followed by 50 GETs out of syncpoint. Channels were defined with BATCHSZ=100 and BATCHINT=120000 (120 seconds) to ensure that the channels did achieve a batch size of 100.

The batch MQI server reply applications simply MQGET-waited on the request queue and returned messages to the reply queue using MQPUT, that is, there was no business application logic. Both MQI calls were outside of syncpoint. These batch application server programs ran non-swappable.
Channel capacity and scalability

If there is message movement between queue managers in a business application then this is often the most significant part of the MQ costs of such an application.

This chapter describes

- The maximum number of channels (or thin clients)
- The factors affecting the throughput and cost of a channel
- The effect of the number of channels on overall throughput and cost per message.
- The effect of the number of channels on channel start and stop rates and costs

What is the maximum number of channels (or thin clients)?

The maximum number of channels is limited by

- Channel initiator virtual storage in the extended private region (EPVT)
  (this applies to all channel types including CHLTYPE(SVRCONN) channels (thin clients))
- Possibly, by achievable channel start (or restart after failure) and stop rates and costs. These increase with the number of channels represented in SYSTEM.CHANNEL.SYNCQ. See “Channel START/STOP rates and costs” on page 77.

Every non-SSL channel uses about 170 KB and every SSL channel about 210KB of extended private region in the channel initiator (CHINIT) address space. Storage is increased if messages larger than 32 KB are being transmitted. This increased storage is freed when

- a sending or thin client channel requires less than half the current buffer size for 10 consecutive sends
- a heartbeat is sent or received

The upper limit is likely to be around 9000 non-SSL or 7500 SSL channels on many systems as EPVT size is unlikely to exceed 1.6GB.

The maximum number of channels (and/or thin clients) where maximum message size is greater than 32KB is of order

\[
\text{EPVTsize(KB)} / (\maxmessage\text{-}size(KB) + 170) \text{ for non-SSL channels} \\
\text{EPVTsize(KB)} / (\maxmessage\text{-}size(KB) + 210) \text{ for SSL channels}
\]

So, for example, if your started channels are all required to move 4000 KB 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.

Also remember that queue managers are often connected by pairs of channels as a channel is required for each direction in which messages are moved.

If you are intending to directly connect desktops to an OS/390 queue manager then remember that each application on the desktop which requires to be an MQ thin client will probably need to connect individually to that queue manager. Each such concurrently run desktop application counts towards your system's limit.
Channel START/STOP rates and costs

The rate and CPU cost at which channels can be started and stopped varies with the number of channels represented in SYSTEM.CHANNEL.SYNCQ. The rate and CPU cost have been successively improved in MQSeries for OS/390 V2.1 and MQSeries for OS/390 V5.2.

While many users do not start and stop channels with any great frequency, there may still be significant sender channel restart activity after a channel initiator failure.

<table>
<thead>
<tr>
<th>Release</th>
<th>Channel pairs in .SYNCQ</th>
<th>Sender Channel START Rate / second 9672-X37 CPU millsecs / START</th>
<th>Sender Channel STOP Rate / second 9672-X37 CPU millsecs / STOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2.1</td>
<td>1,000</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>V2.1</td>
<td>2,000</td>
<td>21</td>
<td>52</td>
</tr>
<tr>
<td>V5.2</td>
<td>1,000</td>
<td>58</td>
<td>17</td>
</tr>
<tr>
<td>V5.2</td>
<td>2,000</td>
<td>52</td>
<td>22</td>
</tr>
<tr>
<td>V5.2</td>
<td>4,000</td>
<td>41</td>
<td>39</td>
</tr>
</tbody>
</table>

A channel is represented in SYSTEM.CHANNEL.SYNCQ if it has ever been started. It will remain represented until its definition is deleted. For this reason we recommend that redundant channel definitions be deleted.

A channel pair is one CHLTYPE(SDR) and one CHLTYPE(RCVR).

Factors affecting channel throughput and cost

- Message persistence, especially for NPMSPEED(FAST) channels
- Message size
- Achieved batch size is very significant for both throughput and cost
  - and is often much smaller than the defined BATCHSZ channel parameter
  - See “Tuning channels - BATCHSZ, BATCHINT, and NPMSPEED” on page 22 which discusses batching and what is meant by 'achieved batch size'.
  - You need to understand what batch size your configuration will typically achieve before using the following charts to estimate possible throughput and CPU cost.
- With a pre-loaded transmission queue you can probably achieve a batch size equal to the BATCHSZ parameter setting.
- Otherwise you can probably only achieve an average batch size < 2 with most batches consisting of just 1 message, unless you can take advantage of the BATCHINT parameter.
- Message throughput is highly dependent on the configuration used.
  - Speed and utilisation of the network
  - Response time of the MQSeries log devices
  - CPU speeds, at both ends
  - Whether messages on the transmission queue have to be retrieved from disk
- MQSeries for OS/390 Version 2.1 with the TCP/IP OE Sockets support available in OS/390 Version 2 Release 5 has performance close to that with APPC. However, OE Sockets support available in OS/390 Version 2 Release 4 does not show the performance benefits over the IUCV TCP/IP interface that subsequent releases do. See SupportPac MP17 for more information on the performance of OE Sockets versus IUCV.
- We have seen no significant further change when using the TCP/IP which is included with OS/390 Version 2 Release 7.
- CPU costs per message transmitted are increased by about 25% where of the order of 100 channel pairs are used and by about 30% where up to 4,000 channel pairs are used. See "Multiple channel cost and throughput scalability" on page 92 where messages are transmitted across a varying number of channels.
- When using large numbers of channels with persistent messages and where low achieved batchesizes are likely specify the CSQ6CHIP ADAPS parameter = 30. Above this any throughput benefits will be small and CPU costs are likely to increase.
Costs of Moving Messages To and From MVS Images

This section considers the total CPU costs of moving messages between queue managers in separate MVS images. 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.

No code page conversion costs are included in any MVS to MVS measurements, see “” on page 11104 for an estimate of typical MQFMT_STRING code page conversion costs.

Figure: Set up for remote put/get measurements

Each MVS image was a 2 engine dedicated LPAR of a 9672-XZ7 (approximately equivalent to a 9672-X27). The MVS systems were CTC connected.

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

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.

![Chart: CPU usage for Put and Send - Non-Persistent Messages](chart.png)

Chart: CPU usage for Put and Send - Non-Persistent Messages
Chart: CPU usage for Receive and Get - Non-Persistent Messages

As with the remote request/reply scenario, the measurements show that, with fewer channel syncpoints, the larger batch size consistently leads to a reduction in the CPU usage at both ends of the transaction. It is possible once again to derive approximate straight line algorithms for each of the measurements shown.

For BATCHSZ(1) the CPU usage is approximately:

At the sender end:

\[(0.85 + 0.015S) \text{ CPU milliseconds per message}\]
where S is the size of the message expressed in 1000’s of bytes

At the receiver end:

\[(0.95 + 0.025S) \text{ CPU milliseconds per message}\]
where S is the size of the message expressed in 1000’s of bytes

e.g. for a 5000 byte message this is:

\[(0.85 + 0.015\times5) = 0.925 \text{ CPU ms at the sender end}\]
\[(0.95 + 0.025\times5) = 1.075 \text{ CPU ms at the receiver end}\]
For BATCHSZ(50) the CPU usage is approximately:

At the sender end:

\((0.55 + 0.015S)\) CPU milliseconds per message

where \(S\) is the size of the message expressed in 1000’s of bytes

At the receiver end:

\((0.75 + 0.022S)\) CPU milliseconds per message

where \(S\) is the size of the message expressed in 1000’s of bytes

e.g. for a 5000 byte message this is:

\((0.55 + 0.015\times5) = 0.625\) CPU ms at the sender end
\((0.75 + 0.022\times5) = 0.86\) CPU ms at the receiver end

These algorithms produce figures which are within 10% of the measured figure for the range shown in the charts.

**Persistent Messages**

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

- BATCHSZ(1) with BATCHINT(0) and one message per driver commit
- BATCHSZ(50) with BATCHINT(0) and one message per driver commit (unconstrained)
- BATCHSZ(50) with BATCHINT(0) and one message per driver commit (constrained)
- BATCHSZ(50) with BATCHINT(0) and fifty messages per driver commit

“Constrained” means that the sender application paused (for 1/50th second) after each put.

The charts below show the total CPU usage in both systems for persistent messages with a variety of message sizes.
Chart: CPU usage for Put and Send - Persistent Messages

Chart: CPU usage for Get - Persistent Messages
Chart: CPU usage for Receive and Get - Persistent Messages

The measurements here show that moving from BATCHSZ(1) to BATCHSZ(50) with an unconstrained driver leads to a reduction in the CPU usage at both ends of the transaction. With the driver application putting messages to the remote queue as fast as it can, the achieved channel batch size is probably very close to the maximum value of 50, which results in a reduction in the number of channel synchronisations and TCPIP transactions per message.

Note that this reduction in CPU is dependent upon messages being put at a suitable rate. In a more realistic situation where the achieved batch size is close to 1, the CPU usage increases, as shown by the measurements with the constrained driver.

Synchronising the channel batch size with the application unit of work size results in a further reduction in CPU usage at both ends due to the reduction in the number of commits and optimisation of the channel synchronisation process.

For capacity planning purposes it is safest to assume BATCHSZ(1) will be used. If it is known that a higher batch size can be achieved, a lower figure may be used. The discussion in the section entitled “Tuning channels - BATCHSZ, BATCHINT, and NPMSPEED” on page 22 indicates that achieving a batch size of 4 can halve the message transmission costs.

Approximate straight line algorithms are presented here for each of the four configurations:
For BATCHSZ(1) the CPU usage is approximately:

<table>
<thead>
<tr>
<th></th>
<th>At the sender end:</th>
<th>At the receiver end:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2.4 + 0.04S) CPU milliseconds per message</td>
<td>(2.4 + 0.04S) CPU milliseconds per message</td>
</tr>
<tr>
<td></td>
<td>where S is the size of the message expressed in 1000’s of bytes</td>
<td>where S is the size of the message expressed in 1000’s of bytes</td>
</tr>
<tr>
<td>e.g. for a 5000 byte message this is:</td>
<td>(2.4 + 0.04*5) = 2.6 CPU ms  at the sender end</td>
<td>(2.0 + 0.04*5) = 2.2 CPU ms  at the receiver end</td>
</tr>
</tbody>
</table>

This represents the ‘best choice scenario’ for persistent messages where the achieved batch size is undetermined.

For BATCHSZ(50) with commit size 1 and an unconstrained driver the CPU usage is approximately:

<table>
<thead>
<tr>
<th></th>
<th>At the sender end:</th>
<th>At the receiver end:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1.5 + 0.035S) CPU milliseconds per message</td>
<td>(1.4 + 0.035S) CPU milliseconds per message</td>
</tr>
<tr>
<td></td>
<td>where S is the size of the message expressed in 1000’s of bytes</td>
<td>where S is the size of the message expressed in 1000’s of bytes</td>
</tr>
<tr>
<td>e.g. for a 5000 byte message this is:</td>
<td>(1.5 + 0.035*5) = 1.675 CPU ms  at the sender end</td>
<td>(1.4 + 0.035*5) = 1.575 CPU ms  at the receiver end</td>
</tr>
</tbody>
</table>

This represents the case where the persistent message rate is sufficient to achieve a batch size of 50.
For BATCHSZ(50) and a low message rate such that the achieved batch size is approximately 1, the CPU usage is approximately:

At the sender end:

\[(3.1 + 0.03S) \text{ CPU milliseconds per message} \]
where S is the size of the message expressed in 1000’s of bytes

At the receiver end:

\[(2.5 + 0.03S) \text{ CPU milliseconds per message} \]
where S is the size of the message expressed in 1000’s of bytes

e.g. for a 5000 byte message this is:

\[
(3.1 + 0.03\times5) = 3.25 \text{ CPU ms at the sender end} \\
(2.5 + 0.03\times5) = 2.65 \text{ CPU ms at the receiver end}
\]

This is most likely to be relevant in a system using the default batch size and a low persistent message rate

For BATCHSZ(50) and a driver commit size of 50 the CPU usage is approximately:

At the sender end:

\[(1.0 + 0.025S) \text{ CPU milliseconds per message} \]
where S is the size of the message expressed in 1000’s of bytes

At the receiver end:

\[(0.75 + 0.03S) \text{ CPU milliseconds per message} \]
where S is the size of the message expressed in 1000’s of bytes

e.g. for a 5000 byte message this is:

\[
(1.0 + 0.025\times5) = 1.125 \text{ CPU ms at the sender end} \\
(0.75 + 0.03\times5) = 0.9 \text{ CPU ms at the receiver end}
\]

This case requires precise synchronisation between the application commit size and the BATCHSZ value, and is unlikely to occur by chance

These algorithms produce figures which are within 10% of the measured figure for the range shown in the charts.
Breakdown of costs

Resources used processing 1000-byte messages

Table: CPU used to process 1000-byte messages

These measurements were taken on a pair of 2 engine dedicated LPARs of a 9672-XZ7 (equivalent to a pair of 9672-X27s)

<table>
<thead>
<tr>
<th>Persistence</th>
<th>N</th>
<th>N</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel batch size (BATCHSZ)</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Messages per sender commit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Achieved Batch Size</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

CPU time used per message pair in CPU milliseconds

| A: Queue manager | 0.001 | 0.001 | 0.6 | 0.7 | 0.4 | 0.04 |
| A: Channel initiator | 0.6 | 0.4 | 1.4 | 2.0 | 0.8 | 0.8 |
| A: TCP/IP | 0.1 | 0.02 | 0.1 | 0.1 | 0.1 | 0.03 |
| A: Application | 0.1 | 0.1 | 0.3 | 0.3 | 0.2 | 0.2 |
| A: Sum of CPU time | 0.8 | 0.5 | 2.4 | 3.1 | 1.5 | 1.0 |

| B: Queue manager | 0.01 | 0.01 | 0.5 | 0.6 | 0.4 | 0.04 |
| B: Channel initiator | 0.7 | 0.5 | 1.0 | 1.4 | 0.7 | 0.5 |
| B: TCP/IP | 0.1 | 0.1 | 0.1 | 0.2 | 0.1 | 0.1 |
| B: Application | 0.2 | 0.2 | 0.2 | 0.3 | 0.2 | 0.1 |
| B: Sum of CPU time | 1.0 | 0.8 | 1.8 | 2.5 | 1.4 | 0.7 |

Resources used processing 10000-byte messages

Table: CPU used to process 10000-byte messages

These measurements were taken on a pair of 2 engine dedicated LPARs of a 9672-XZ7 (equivalent to a pair of 9672-X27s)

<table>
<thead>
<tr>
<th>Persistence</th>
<th>N</th>
<th>N</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel batch size (BATCHSZ)</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Messages per sender commit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Achieved Batch Size</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

CPU time used per message pair in CPU milliseconds

| A: Queue manager | 0.01 | 0.01 | 0.8 | 0.9 | 0.5 | 0.1 |
| A: Channel initiator | 1.6 | 1.2 | 1.6 | 2.0 | 1.0 | 0.7 |
| A: TCP/IP | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| A: Application | 0.5 | 0.5 | 0.3 | 0.4 | 0.3 | 0.3 |
| A: Sum of CPU time | 2.2 | 1.8 | 2.8 | 3.4 | 1.9 | 1.2 |
| B: Queue manager | 0.1 | 0.1 | 0.7 | 0.7 | 0.4 | 0.1 |
| B: Channel initiator | 1.4 | 1.1 | 1.3 | 1.6 | 1.0 | 0.8 |
| B: TCP/IP | 0.1 | 0.1 | 0.1 | 0.2 | 0.1 | 0.1 |
| B: Application | 0.6 | 0.6 | 0.4 | 0.4 | 0.3 | 0.2 |
| B: Sum of CPU time | 2.2 | 1.9 | 2.5 | 2.9 | 1.8 | 1.2 |
Single channel costs - making sense of the data

The following charts give message throughput capacity (messages/sec) and CPU costs (CPUmillisecs/message) for transmitting messages from a MVS sender queue manager to a separate MVS receiver queue manager. These were measured with a single sender channel to a single receiver channel both defined NPMSPEED(FAST). The MVS systems were CTC connected.

These results were measured on MQSeries for OS/390 Version 2.1 which showed a CPU cost improvement of between 2% and 7% versus Version 1.2.

Charting CPU costs against the number of batches per 100 msgs, that is 100 / achieved_batch_size, shows a reasonably straight line. This is consistent with a fixed base cost per message (of given persistence and length) plus a fixed cost per completed batch which is spread over the number of messages in the achieved batch size.

It is therefore reasonable to interpolate costs for other achieved batch sizes. For example consider the chart for persistent messages in the 10K on page 89

- if achieved batch size is expected to be about 4 then this equates to 100/4 (= 25) batches / 100 msgs.
- we can now read off of the chart a value of about 2.6 9672-R55 CPUmillisecs / msg for the cost of receiving one 10K persistent message at 25 batches per 100 msgs.
- Note that an achieved batch size of just 2 (50 batches per 100 msgs) gives 50% of any savings possible on the costs at achieved batch size 1 (100 batches per 100 msgs).
CPU costs using TCP for transmitting persistent messages

- V2.1 1K Pers - Sender
- V2.1 1K Pers - Receiver
- V2.1 10K Pers - Sender
- V2.1 10K Pers - Receiver

CPU costs using TCP for transmitting nonpersistent messages

- V2.1 1K Nonp - Sender
- V2.1 1K Nonp - Receiver
- V2.1 10K Nonp - Sender
- V2.1 10K Nonp - Receiver

Figure: V2.1 Single Channel CPU Costs - Persistent Messages - TCP OESOCKETS

Figure: V2.1 Single NPMSPEED(FAST) Channel CPU Costs - Nonpersistent Msgs - TCP OESOCKETS
The APPC definitions for the listeners used the standard #INTER logmode. The VTAM definition for the PU for the CTC between the systems specified DELAY=0. The RUSIZE was 256. RUSIZE=3480 is probably more appropriate for 10K messages (see on page 46.)
Figure: V2.1 Single Channel Capacity - Achieved Batchsize Effects

Note that

- a high achieved batchsize (low number of batches per 100 msgs) gives much greater message rates.
- the data rate (size of message × number of messages per sec) is much higher for 10K than for 1K messages.
Multiple channel cost and throughput scalability

The following charts were derived from measurements using a varying number of channels between just two queue managers connected via a single TCP/IP link. All measurements are from a set of sending channels on one system to a set of receiving channels on another. That is, all messages were transmitted in only one direction. There was no further processing of any of the received messages.

Two different sets of measurements are included.

- The first set are for two CTC connected 9672-R55’s running OS/390 Version 2 Release 5. See “Figure: V2.1 Multiple BATCHSZ(1) Channels - Nonpersistent Message Cost Scalability” on page 93 and “Figure: V2.1 BATCHSZ(1) Channels on CTC Link - Nonp Message Rate Scalability” on page 96
  - These measurements are all for 1K byte nonpersistent messages being generated as quickly as possible by multiple batch applications on the sender queue manager which are transmitted by pre-started channels to a receiver queue manager. That is the queues are not pre-loaded for these measurements. This allowed up to 2000 channels to be used concurrently without filling any bufferpools.
  - The channels are defined with BATCHSZ(1) and NPMSPEED(FAST). BATCHSZ(1) is used as this is the best way to achieve a consistent batch size for every measurement without using pre-loaded queues.
  - All measurements used CSQ6CHIP parameters ADAPS=16 and DISPS=50.
- The second set are for a 9672-X37 connected via OSA Express->ATM(155Mbit)->OSA Express to another 9672-X37. The OSAs were specified as token ring. Both ends were running OS/390 Version 2 Release 10. In this set MQSeries for OS/390 Version 5.2 was measured.
  - 1000 byte persistent, 1000 and 10,000 byte nonpersistent messages were used.
  - Pre-loaded transmission queues were used.
    - thus allowing any batchsize choice to be measured
    - and also allowing more precision than the first measurement set methodology, but only up to 128 channels. Higher numbers of channels would eventually fill bufferpools and thus significantly alter the cost per message.
  - NPMSPEED(FAST) and DISCONN(1) were used for all measurements.
  - BATCHSZ(1) and BATCHSZ(50) measurements were taken for both persistent and nonpersistent messages.
  - All measurements used CSQ6CHIP parameters ADAPS=30 and DISPS=20 except for a further measurement of persistent messages at BATCHSZ(1) which used ADAPS=15.
  - Channel start and disconnect costs for zero messages were separately taken and subtracted from measurements to give the results shown.
Message cost scalability

The following chart shows that the (9672-R55) CPU cost per message remains within about 125% of the cost of transmitting over a single channel. At 2,000 channels the cost per message was still within 130% of that for a single channel. This result was found using the first measurement set method which tends to exaggerate the cost for large numbers of channels. This is because with large numbers of channels it is much more likely that attempts to get messages from empty transmission queues are made.

The following charts show the cost per message in 9672-X37 CPU milliseconds at both the sending and receiving end of the channels using the second measurement method.

The second measurement set method gives a purer cost for moving messages over channels because there is no other MQSeries activity going on when using pre-loaded queues. This allows the queue manager address space costs to be included which is needed for persistent message measurement. However the queue manager address space cost is typically less than 5% of the total costs for nonpersistent messages (on NPMSPEED(FAST) channels).
Figure: V5.2 Multiple Channels - 1000 byte Nonpersistent Message CPU Cost Scalability. OSA Express -> ATM(155Mbit) -> OSA Express link

Figure: V5.2 Multiple Channels - 10,000 byte Nonpersistent Message CPU Cost Scalability. OSA Express -> ATM(155Mbit) -> OSA Express link
Figure: V5.2 Multiple Channels - 1000 byte Persistent Message Receiver Channel CPU Cost Scalability. OSA Express -> ATM(155Mbit) -> OSA Express link OSA->ATM(155Mbit)-->OSA link
Message throughput scalability

Message rates depend on the type of link used as well as on MQSeries parameters like BATCHSZ and message persistence.

For the first measurement set which used the CTC link the total nonpersistent message rate is limited only by the power of the CPU (a 9672-R55). More than 2100 messages/sec was achieved with 2000 channel pairs.

![Message Rate Scalability - Single TCP/IP Link](image)

For BATCHSZ(1) channels with persistent messages increasing the number of adapters (ADAPS= in CSQ6CHIP) can increase message throughput. It is not recommended to use a value higher than about 30. Throughput approaches the maximum at about 30 while CPU costs increase for large numbers of channels as the ADAPS value increases.

In the second measurement set (using a 9672-X37 system), the results of which are shown below, maximum throughput was generally constrained by the OSA Express->ATM(155Mbit)->OSA Express bandwidth, although for BATCHSZ(50) with 16, 64 and 128 channels receiver end total CPU exceeded 90%.

It should be noted that the methodology used for this set of measurements tends to produce conservative values for maximum throughput with larger numbers of concurrent channels. The measurements involved pre-loading transmission queues, starting channels simultaneously then waiting for the last channel to disconnect. However in reality the channels did not start simultaneously: There was a spread of about 2 seconds between the 1st and 64th channel start, and about 4 seconds between the 1st and 128th channel start. At the higher throughput rates, all pre-loaded messages were sent in less than 20 seconds; throughput was calculated by just subtracting the channel DISCINT value of 1 seconds from this elapsed time.
Figure: V5.2 Multiple Channels - 1000 byte Nonpersistent Message Throughput Scalability. OSA Express -> ATM(155Mbit) -> OSA Express link

Figure: V5.2 Multiple Channels - 10,000 byte Nonpersistent Message Throughput Scalability. OSA Express -> ATM(155Mbit) -> OSA Express link
Figure: V5.2 Multiple Channels - 1000 byte Persistent Message Throughput Scalability.
OSA Express -> ATM(155Mbit) -> OSA Express link

The first 3 points for Batchsz 1, are approx 67, 198, 513 for 1, 4, 16 channels respectively.
Some basic MQSeries API call costs (for local, non-shared queues)

The following section has some ‘rules of thumb’ regarding how much CPU different MQI verbs use. Before using the rules you should note the following:

1. The rules generally give the lowest cost - it may be larger in practice.
2. If doing capacity planning estimates you should allow an additional 20%.
3. The measurements were taken with accounting class(1) trace running. Turning extra traces on will increase the CPU used.

This data might help to bridge from a specific application example to your own case with particular message sizes and the odd extra MQPUT or MQGET. You should try and start from an application example level rather than this low level.


If you are considering using very large messages (those greater than 4 MB) then please also read “on page 179.

How big a message fits within a single page?

In the following algorithms messages which fit within a single page are those which are \( \leq 3785 \) bytes including all headers except the MQMD (\( \leq 4009 \) bytes including the MQMD).

MQGET and MQPUT

The data presented here provides approximate algorithms for calculating the cost of MQPUT or MQGET calls. The algorithms have been derived from measured data for batch jobs running under OS/390 V2.10 on a 3 dedicated engine LPAR of a 9672-XZ7 (near equivalent to a 9672-X37) using MQSeries for OS/390 V5.2.

These batch applications use queues which do not fill their bufferpools. There is no pageset I/O in any of these measurements. All the queues are local, non-shared queues.

The CPU measurements used in deriving these algorithms include the costs of MQGET, MQPUT and, MQCMIT where relevant, but exclude the costs of MQOPEN and MQCLOSE.

Non-indexed queue MQGET and MQPUT call costs are not significantly different between Versions 5.2, 2.1, and 1.2 for capacity planning purposes.

Indexed queue MQGET and MQPUT call costs are typically similar to those for non-indexed queues for V2.1 and later. An indexed queue with very large numbers of messages will have greater costs than a small queue. Version 2.1 indexed queue support has significant CPU cost improvements over V1.2:

• Where queue depth is large for MQPUT calls with the same MsgId (or CorrelId)
• For MQGET calls where all messages are of the same priority
• For MQGET calls with browse
Most applications that use indexed queues do so in a way such that CURDEPTH is never large. In these cases V1.2 indexed queue costs will also be similar to V1.2 non-indexed queue costs.

There are reasonably small step differences in cost as the message size (plus header) increases. These steps are associated with the number of 4K pages required to hold a message. For instance two messages of user data size 1000 bytes can share one page, while a 5000 byte message requires two pages of its own. (Messages larger than 1 page do not share any partially used page.)

Message length in the following algorithms means user message length plus all headers except the MQMD. The MQMD is always present, all other headers, for instance the MQXQH (transmission queue header), are required or not according to application usage or queue definition.

Summary:

a. For messages up to about 10,000 bytes in length, batching several messages into a single unit of work generally gives better performance than handling messages singly.
   - For up to about 100 messages per unit of work.
   - The optimum number of messages per unit of work is around 50.

b. It is more efficient to get non-persistent messages out of syncpoint - but business logic may require the gets to be done within syncpoint

c. It is cheaper to use MQPUT1 when there is only on message per transaction; if more than one message is put to a queue it is cheaper to issue MQOPEN MQPUT MQPUT .. MQCLOSE

Non persistent messages - within syncpoint

For reasonable batch sizes (up to about 100 messages per commit for messages up to about 10KB) the cost of each unit of work:

- is dependent upon the size of the messages
- Increases approximately linearly with the number of messages in the batch
- varies according to the number of pages required per message

CPU costs for MQGET conform to the following algorithms (where S is the size of the message in bytes (including all headers except the MQMD) and there are n messages per unit of work):

<table>
<thead>
<tr>
<th>Scenario</th>
<th>CPU Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>For messages within a single page,</td>
<td>$CPU cost = (95 + 0.015S) + n(70 + 0.005S)$ (microseconds)</td>
</tr>
<tr>
<td>For messages greater than a single page,</td>
<td>$CPU cost = 95 + n(115 + 0.0045*S)$ (microseconds)</td>
</tr>
</tbody>
</table>

CPU costs for MQPUT conform to the following algorithms (where S is the size of the message in bytes (including all headers except the MQMD) and there are n messages per unit of work):
For messages within a single page,

\[ \text{CPU cost} = (105 + .001S) + n(85 + 0.024S) \] (microseconds)

For messages greater than a single page,

\[ \text{CPU cost} = 115 + n(120 + 0.0075S) \] (microseconds)

The algorithms detailed above are valid for messages of up to 1 Megabyte. The CPU cost of messages greater than 1 Megabyte exhibits a small degree of non-linearity, with the ‘cost per byte’ slowly increasing with message size.

The CPU costs for messages in the range 1 - 100 Megabytes approximately conform to the following algorithms (where \( M \) is the size of the message in Megabytes):

- For MQPUT, cost = \( 1200 + 7000*(M^{1.09}) \) (microseconds)
- For MQGET, cost = \( 2000 + 3000*(M^{1.23}) \) (microseconds)

**EXAMPLES:**

MQGET 32 non-persistent messages of 1000 bytes.

\[
\text{Total cost} = 95 + 0.015*1000 + (32)(70 + 0.005*1000)
\]

\[
= 2510 \text{ CPU microseconds}
\]

MQPUT 16 non-persistent messages of 5000 bytes.

\[
\text{Total cost} = 115 + (16)(120 + 0.0075*5000)
\]

\[
= 2635 \text{ CPU microseconds}
\]

For units of work comprising up to 32 messages, the incremental algorithms are valid for message sizes up to 100,000 bytes. For units of work comprising up to 64 messages, they are valid for message sizes up to 50,000 bytes. For larger messages very low numbers of messages per commit are recommended.

**Persistent messages - within syncpoint**

The CPU costs for MQPUTs and MQGETs of persistent messages are higher than the costs for non-persistent messages. But they do follow the same general trend, conforming to the following algorithms (where \( S \) is the size of the message in bytes (including all headers except the MQMD) and there are \( n \) messages per unit of work):
For MQGET:

For messages within a single page,
\[ \text{CPU cost} = (285 + 0.001S) + n(85 + 0.004S) \] (microseconds)

For messages greater than a single page,
\[ \text{CPU cost} = 275 + n(125 + 0.0055*S) \] (microseconds)

For MQPUT:

For messages within a single page,
\[ \text{CPU cost} = (285 + 0.033S) + n(100 + 0.042S) \] (microseconds)

For messages greater than a single page,
\[ \text{CPU cost} = 410 + n(150 + 0.014*S) \] (microseconds)

Large numbers of message operations per commit are increasingly expensive per operation, we recommend you stay within about 50 message operations per commit

**EXAMPLES:**

MQGET 32 persistent messages of 1000 bytes.
Total cost \[ = 285 + 0.001*1000 + (32)(85 + 0.004*1000) \]
\[ = 3134 \text{ CPU microseconds} \]

MQPUT 16 persistent messages of 5000 bytes.
Total cost \[ = 410 + (16)(150 + 0.014*5000) \]
\[ = 3930 \text{ CPU microseconds} \]

**Non persistent messages - out of syncpoint**

The CPU cost for MQGET or MQPUT of a single message out of syncpoint is less than that of a single message within syncpoint.

For non-persistent messages the CPU costs of a single message out of syncpoint conform to the following algorithms (where S is the size of the message in bytes (including all headers except the MQMD) :
For MQGET:

For messages within a single page,

CPU cost = (70 + 0.005S) (microseconds)

For messages greater than a single page,

CPU cost = (112 + 0.0042*S) (microseconds)

For MQPUT:

For messages within a single page,

CPU cost = (80 + 0.023S) (microseconds)

For messages greater than a single page,

CPU cost = (115 + 0.0067*S) (microseconds)

**EXAMPLES:**

MQGET 1 non-persistent messages of 1000 bytes out of syncpoint.

Total cost = 70 + 0.005*\textbf{1000}

= 75 CPU microseconds (9672-X37)

MQPUT 1 non-persistent messages of 5000 bytes out of syncpoint.

Total cost = 115 + 0.0067*\textbf{5000}

= 150 CPU microseconds (9672-X37)

As a rule of thumb, the cost difference between a non-persistent message within syncpoint and the same message out of syncpoint is in the range 100 - 125 CPU microseconds (9672-X37).

**Persistent messages - out of syncpoint**

For capacity planning purposes there is no significant difference between an out of syncpoint MQGET or MQPUT and a within syncpoint single MQGET or single MQPUT of a persistent message.

**How much extra does each waiting MQGET cost?**

If an MQGET has to wait for a message then there is an additional cost of about 110 CPUmicrosecs (9672-X37) for each wait.

This cost is not dependent on message length, persistence, or whether in or out of syncpoint.

If you have more than one application waiting for a message on a particular queue then every such application will race for any newly arriving message. Only one application will win this race, the other applications will have to wait again. So if you have, for example, five applications all waiting for a message on a particular queue the total cost to get the message is the cost of a successful MQGET as in the tables above (which, of course, does depend on message length, persistence, and whether in or out of syncpoint) plus 5 times 110 CPUmicrosecs (9672-X37).
How much extra does code page conversion cost on an MQGET?

Code page conversion from one single byte character set to another using MQFMT_STRING costs about the same as a basic MQGET out of syncpoint for the same message size.

DBCS to DBCS conversion costs are of order 4 times a basic MQGET out of syncpoint for the same message size.

MQCONN/MQDISC

The cost for a batch application to connect and disconnect from MQSeries is approximately 3100 CPUmicroseconds (9672-X37).

MQOPEN/MQCLOSE (inc MQPUT1) - non-dynamic queues

The cost to repeatedly and explicitly MQOPEN plus MQCLOSE a non-dynamic queue for output is about 110 CPUmicroseconds (9672-X37). MQPUT1 is cheaper than an explicit MQOPEN/MQCLOSE for output where only a single message is put. MQPUT1 costs about 65 CPUmicroseconds (9672-X37) more than an MQPUT for the same message.

The cost to repeatedly and explicitly MQOPEN plus MQCLOSE a non-dynamic queue for input is about 155 CPUmicroseconds (9672-X37).

Dynamic queue creation

The CPU cost of creating many dynamic queues is greatly reduced in Version 2.1. The cost now increases slowly as the number of dynamic queues already defined increases, and is about 1300 CPUmicroseconds (9672-X37) to create at MQOPEN (plus MQCLOSE) each queue (measured with up to 3000 permanent dynamic queues). In Version 1.2, there is a significant increase in cost as the number of queues defined increases, because of a scan for the new queue being an initiation queue for another. The average cost for each queue is about 6300 CPUmicroseconds where there are 1000 such queues created.

What is the cost of creating a trigger or event message?

The cost of creating one trigger message is approximately the same as MQPUT of a small non-persistent message (less than 100 bytes).

The cost of creating each event message is approximately the same as MQPUT of a small persistent message.

Combining requests

To estimate the resources used when MQPUTS and MQGETS are used in the same unit of work, you can just add the information used above.

For example application-1 which does

- Put Put Commit
• Get (with wait) Get (with wait) Commit

And a partner application-2 which does
• Get Put Get Put Commit

Messages are 1000 byte non-persistent messages

Application 1

MQPUT CPU = \(105 + 0.025S + n(85 + 0.0024S)\)
\[= 105 + 25 + 2*(85 + 2.4)\]
\[= 304\ \text{microseconds (9672-X37)}\]

MQGET CPU = \(95 + 0.015S + n(70 + 0.005S)\)
\[= 95 + 15 + 2*(70 + 5)\]
\[= 260\ \text{microseconds (9672-X37)}\]

The application had to wait for the first message, so this is an additional 110 microseconds

Total estimated cost = 304 + 260 + 110 = 674 microsecs (9672-X37)

Measured average cost over 1000 loops = 719 microseconds

Application 2 is the same as application 1, but there is only one commit instead of 2 commits. Ignore this for estimation purposes.

Estimate of application 2 costs = 674

Measured average cost over 1000 loops = 670

For application 1 + an (open + close) for both queues.

CPU = application 1 above + 2 (open + close)
\[= 674 + 110 + 100\]
\[= 894\ \text{microsecs (9672-X37)}\]

Measured time over 1000 loops = 890

---

**Trace Costs**

**Global Trace Costs**

An internal trace, TRACE(G) DEST(RES), increases CPU cost by the order of 60% to 80%.

**Accounting Trace Costs**

TRACE(A) CLASS(1) costs were part of the measurements for this chapter. They are of the order of an extra 2% to 3% over having no accounting trace.
TRACE(A) CLASS(3), first available with MQSeries for OS/390 V5.2, costs an additional 5% to 10%.

**Statistics Trace Costs**
TRACEx(S) costs are not significant.

---

**Capacity planning example**

**Scenario**
Application 1 puts a 500 byte message using MQPUT1 out of syncpoint
Batch Application 2 gets a number of messages and commits.
1 Million messages per 10 hour shift (28/second)

**Application 1**
Cost per message is $70 + 0.005S + 65 = 137.5$ CPU microsecs (9672-X37)

**Application 2**
Cost per message is $285 + 0.001S + n*(85 + 0.004S)$

$$= 285.5 + n*87$$

For 1 Million messages.
Application 1 = $137.5 \times 1,000,000$ microseconds = 137.5 seconds
Application 2, 1 message per commit $(285.5 + 87) \times 1,000,000$ microseconds = 372.5 seconds
Application 2, 2 message per commit $(285.5 + 2\times87) \times 1,000,000/2$ microseconds = 229.8 seconds
Application 2, 50 message per commit $(285.5 + 50\times87) \times 1,000,000/50$ microseconds = 92.7 seconds
Total CPU = $137.5 + 372.5$ (Worst case) = 510 seconds
Allow an extra 10-20%, CPU needed = 560 - 610 seconds of CPU of a 9672-X37

If Application 1 could put more messages within a unit of work the CPU cost would drop in a similar way to the Get cost.
Capacity planning for a requester/reply model with remote messages

This section considers the costs of request and reply messages using remote queuing. The scenario, described in detail below, involves two different MVS images. A requester application attached to a queue manager running in System A puts a message to a 'request' queue which is defined on a queue manager running in System B (i.e. a remote queue), and requests a reply message to be placed on a named queue in the initiating queue manager. The messages are moved between the two systems using TCP/IP. Having put a message, the requester application does a 'get with wait' on its local reply queue. A reply application in System B gets any messages from its local 'request' queue and puts an identical reply message to the named reply queue.

Measurement scenario
A simple remote queue manager driven requester/reply scenario was used. Up to 10 requester queue managers (drivers) were started on one MVS image and a single reply queue manager (server) was started in a separate MVS image. Each MVS image was a 2 engine dedicated LPAR of a 9672-XZ7 (approximately equivalent to a 9672-X27). All of the queue managers were defined as belonging to the same MQ cluster, with each queue manager having a single CLUSRCVR channel. The ‘target’ queue was defined to the cluster, but the reply queues (one per client queue manager) were not. The reply application determines the destination of the reply message from the MQMD of the requester message. This set up results in a single sender and receiver channel pair between each requester queue manager and the server queue manager. The MVS systems were CTC connected.

The driver application puts a message to the request queue and then issues a get-with-wait against the reply queue.

The server application continually loops, using get-with-wait to retrieve a message, followed by an MQPUT1 for the reply message and then a commit. All server calls are done within syncpoint.

The driver application retrieves the reply message and then puts the next message to the reply queue.

Neither application involves any business logic.

The server application runs non-swappable.

The request queue is not indexed.

The reply queues are indexed on MSGID. Each requester message has a unique msgid, which is copied to the reply message. The driver application uses this msgid in retrieving the reply message.

The throughput achieved with this set up may be limited by either the speed of the network, the amount of CPU power available or (for persistent messages) the log data rate. In some instances the throughput can be increased by increasing the number of driver and/or server applications. For 1000-byte non-persistent messages we found that two driver applications per driver queue manager (20 driver applications in total) and a single replier application would sustain close to 1000 round trip message pairs per second, while using close to 100% of both LPARs. The same configuration (20 drivers and one replier) was used for all measurements, although other configurations may result in higher throughput rates.

**Measured Results**

**Non-Persistent Messages**

All non-persistent messages were put out of syncpoint by the driver applications. Measurements of non-persistent messages were made with two different channel settings:

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

The BATCHSZ setting determines the maximum number of messages which can be sent before a channel syncpoint occurs. With BATCHSZ(1), a syncpoint occurs with every message; with BATCHSZ(50) with BATCHINT(1000), a syncpoint occurs after 50 messages or 1000 milliseconds after the first message, whichever occurs first. NPMSPEED(FAST) means that non-persistent messages are eligible to be retrieved from the target queue as soon as they arrive, without waiting for the syncpoint to occur. The major difference between the two measurements is the syncpoint frequency.

The charts below show the total CPU usage per transaction (application plus queue manager plus channel initiator plus TCPIP) in each system for a variety of message sizes.
Chart: CPU usage for Requester System - Non-Persistent Messages
The measurements show that the larger batch size consistently leads to a reduction in the CPU usage in both the driver and the replier systems. Although none of the measurements produces a straight line graph, it is possible to derive approximate straight line algorithms for each of the measurements shown.

For BATCHSZ(1) the CPU usage at either the driver or the replier system is approximately:

\[(1.75 + 0.06S) \text{ CPU milliseconds per transaction}\]

where \(S\) is the size of the message expressed in 1000’s of bytes

e.g. for a 5000 byte message this is \((1.75 + 0.06*5) = 2.05\) CPU ms

For BATCHSZ(50) the CPU usage at either the driver or the replier system is approximately:

\[(1.5 + 0.05S) \text{ CPU milliseconds per transaction}\]

where \(S\) is the size of the message expressed in 1000’s of bytes

e.g. for a 5000 byte message this is \((1.5 + 0.05*5) = 1.75\) CPU ms

These algorithms produce figures which are within 10% of the measured figure for the range shown in the charts. Note however that it is not safe to extrapolate the algorithms to larger message sizes. The charts exhibit a slight upward curve in all cases. It is likely that the straight line algorithm will underestimate the CPU cost of transactions with larger message sizes.
Persistent Messages

All persistent messages were put within syncpoint by the driver applications. Persistent message measurements were made with three different configurations:

- BATCHSZ(1) with BATCHINT(0) and one message per driver commit
- BATCHSZ(50) with BATCHINT(0) and one message per driver commit
- BATCHSZ(50) with BATCHINT(0) and fifty messages per driver commit

Persistent messages must always wait for a channel syncpoint before they are eligible for retrieval from the target queue. Any BATCHINT value greater than zero is likely to result in a delay before the channel syncpoint and thus introduce a delay before the messages are available at the remote end.

The charts below show the total CPU usage per transaction in each system for persistent messages with a variety of message sizes.

![Chart: CPU usage for Requester System - Persistent Messages](chart.png)
The measurements show that, for persistent messages, moving from BATCHSZ(1) to BATCHSZ(50) consistently leads to an increase in the CPU usage in both the driver and the replier systems. In both cases the achieved batch size is likely to be 1, although with two applications using the same channel, and BATCHSZ(50), an achieved batch size of 2 would be possible. The increase in CPU usage results from a difference in the way that the channel syncpoint is handled. With BATCHSZ(1) the message and the end of batch indicator are sent in a single flow; with BATCHSZ(50) the message and the end of batch indicator are sent in separate flows, leading to increased costs in both TCP/IP and the channel initiator.

When the channel batch size is synchronised with the driver application unit of work size (i.e. both set to 50) the CPU usage in both the driver and the replier systems is reduced. This is because there are fewer commits, fewer channel synchronisations and fewer TCP/IP transactions per message.

In our scenario the replier treats each message separately (GET-PUT1-COMMIT). Consequently a batch size of 50 is not achieved for the reply message. The DISPLAY CHSTATUS command for the reply channel showed an achieved batch size of between 3 and 4. The discussion in the section entitled “Tuning channels - BATCHSZ, BATCHINT, and NPMSPEED” on page 22 indicates that achieving a batch size of 4 can halve the message transmission costs.

Once again it is possible to derive approximate straight line algorithms for the measurements shown. These algorithms produce figures which are within 10% of the measured figure for the range shown in the following.
For BATCHSZ(1) the CPU usage is approximately:

\[
\begin{align*}
\text{At the driver end:} & \\
(4.8 + 0.12S) \text{ CPU milliseconds per transaction} \\
\text{where } S \text{ is the size of the message expressed in 1000's of bytes} \\
\text{At the replier end:} & \\
(3.7 + 0.12S) \text{ CPU milliseconds per transaction} \\
\text{where } S \text{ is the size of the message expressed in 1000's of bytes} \\
\text{e.g. for a 5000 byte message this is:} & \\
(4.8 + 0.12\times5) &= 5.4 \text{ CPU ms at the driver end} \\
(3.7 + 0.12\times5) &= 4.3 \text{ CPU ms at the replier end}
\end{align*}
\]

For BATCHSZ(50) with an achieved batch size of 1 on the replier’s receiving channels the CPU usage is approximately:

\[
\begin{align*}
\text{At the driver end:} & \\
(5.7 + 0.09) \text{ CPU milliseconds per transaction} \\
\text{where } S \text{ is the size of the message expressed in 1000’s of bytes} \\
\text{At the replier end:} & \\
(4.4 + 0.12S) \text{ CPU milliseconds per transaction} \\
\text{where } S \text{ is the size of the message expressed in 1000’s of bytes} \\
\text{e.g. for a 5000 byte message this is:} & \\
(5.7 + 0.09\times5) &= 6.15 \text{ CPU ms at the driver end} \\
(4.4 + 0.12\times5) &= 5.0 \text{ CPU ms at the replier end}
\end{align*}
\]

This is the cost most likely to be relevant in a typical untuned system.

For BATCHSZ(50) with an achieved batch size of 50 on the replier’s receiving channels the CPU usage is approximately:

\[
\begin{align*}
\text{At the driver end:} & \\
(2.1 + 0.08S) \text{ CPU milliseconds per transaction} \\
\text{where } S \text{ is the size of the message expressed in 1000’s of bytes} \\
\text{At the replier end:} & \\
(2.2 + 0.1S) \text{ CPU milliseconds per transaction} \\
\text{where } S \text{ is the size of the message expressed in 1000’s of bytes} \\
\text{e.g. for a 5000 byte message this is:} & \\
(2.1 + 0.08\times5) &= 2.5 \text{ CPU ms at the driver end} \\
(2.2 + 0.1\times5) &= 2.7 \text{ CPU ms at the replier end}
\end{align*}
\]

This is probably the best case scenario.
### Resources used processing 1000-byte messages

<table>
<thead>
<tr>
<th>Persistence</th>
<th>N</th>
<th>N</th>
<th>P</th>
<th>P</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel batch size (BATCHSZ)</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Messages per driver commit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Round trip message pairs per second</td>
<td>856</td>
<td>978</td>
<td>161</td>
<td>165</td>
<td>195</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU time used per message pair in CPU milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Queue manager</td>
</tr>
<tr>
<td>A: Channel initiator</td>
</tr>
<tr>
<td>A: TCP/IP</td>
</tr>
<tr>
<td>A: Application</td>
</tr>
<tr>
<td>A: Sum of CPU time</td>
</tr>
<tr>
<td>B: Queue manager</td>
</tr>
<tr>
<td>B: Channel initiator</td>
</tr>
<tr>
<td>B: TCP/IP</td>
</tr>
<tr>
<td>B: Application</td>
</tr>
<tr>
<td>B: Sum of CPU time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total CPU used in MVS systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Average CPU % busy</td>
</tr>
<tr>
<td>B: Average CPU % busy</td>
</tr>
</tbody>
</table>

### Resources used processing 10000-byte messages

<table>
<thead>
<tr>
<th>Persistence</th>
<th>N</th>
<th>N</th>
<th>P</th>
<th>P</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel batch size (BATCHSZ)</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Messages per driver commit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Round trip message pairs per second</td>
<td>747</td>
<td>893</td>
<td>107</td>
<td>110</td>
<td>116</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU time used per message pair in CPU milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Queue manager</td>
</tr>
<tr>
<td>A: Channel initiator</td>
</tr>
<tr>
<td>A: TCP/IP</td>
</tr>
<tr>
<td>A: Application</td>
</tr>
<tr>
<td>A: Sum of CPU time</td>
</tr>
<tr>
<td>B: Queue manager</td>
</tr>
<tr>
<td>B: Channel initiator</td>
</tr>
<tr>
<td>B: TCP/IP</td>
</tr>
<tr>
<td>B: Application</td>
</tr>
<tr>
<td>B: Sum of CPU time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total CPU used in MVS systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Average CPU % busy</td>
</tr>
</tbody>
</table>
Capacity planning for a CICS requester / Batch reply model with local messages

This chapter compares the CPU costs associated with putting and getting messages of different size and persistence. It also describes how the transaction rate affects the performance of your system.

The test scenario

The application scenario and system configuration used to take the measurements described in this chapter is the same as that described in “The test scenario” on page 27 with the following differences:

- The queues were not preloaded with messages.
- Four server queues were used with six servers per server queue (a total of 24 server programs).

Summary of results

This section gives a summary of the CPU used when running this application using messages of different sizes and persistence. It also gives a summary of the maximum transaction rates achieved. The CPU cost per message is not affected much by the transaction rate.

<table>
<thead>
<tr>
<th>Message size in bytes</th>
<th>Persistence</th>
<th>Cost per transaction in service units</th>
<th>Cost per transaction in milliseconds of CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000</td>
<td>Persistent</td>
<td>16.0</td>
<td>8.4</td>
</tr>
<tr>
<td>1 000</td>
<td>Nonpersistent</td>
<td>13.1</td>
<td>6.9</td>
</tr>
<tr>
<td>1 000</td>
<td>Nonpersistent 1</td>
<td>7.1</td>
<td>3.7</td>
</tr>
<tr>
<td>10 000</td>
<td>Persistent</td>
<td>24.0</td>
<td>12.6</td>
</tr>
<tr>
<td>10 000</td>
<td>Nonpersistent</td>
<td>14.5</td>
<td>7.6</td>
</tr>
<tr>
<td>100 000</td>
<td>Persistent</td>
<td>57.6</td>
<td>30.2</td>
</tr>
<tr>
<td>100 000</td>
<td>Nonpersistent</td>
<td>50.7</td>
<td>26.6</td>
</tr>
</tbody>
</table>

Note.

1. This test was done with four server queues and six server programs per server queue.
2. This test was done with four server queues and one server program per server queue. Compared to the test when six server programs per server queue were used, one server program per server queue had a higher throughput, and used almost half the CPU.
Limiting factors:

1. CPU within the processor
2. The rate TPNS could drive transactions at a high enough rate
3. Log I/O
4. Internal locking

<table>
<thead>
<tr>
<th>Message size in bytes</th>
<th>Persistence</th>
<th>Transactions per hour</th>
<th>Messages per hour</th>
<th>Limiting factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000</td>
<td>Persistent</td>
<td>1 080 000</td>
<td>2 160 000</td>
<td>1</td>
</tr>
<tr>
<td>1 000 Nonpersistent</td>
<td>(6 server programs per server queue)</td>
<td>1 013 760</td>
<td>2 027 520</td>
<td>1</td>
</tr>
<tr>
<td>1 000 Nonpersistent</td>
<td>(1 server program per server queue)</td>
<td>1 205 000</td>
<td>2 410 000</td>
<td>2</td>
</tr>
<tr>
<td>10 000 Persistent</td>
<td></td>
<td>347 000</td>
<td>694 000</td>
<td>3</td>
</tr>
<tr>
<td>10 000 Nonpersistent</td>
<td></td>
<td>756 000</td>
<td>1 512 000</td>
<td>4</td>
</tr>
<tr>
<td>100 000 Persistent</td>
<td></td>
<td>38 000</td>
<td>76 000</td>
<td>3</td>
</tr>
<tr>
<td>100 000 Nonpersistent</td>
<td></td>
<td>340 000</td>
<td>680 000</td>
<td>4</td>
</tr>
</tbody>
</table>

Limiting factors:

1. CPU within the processor
2. The rate TPNS could drive transactions at a high enough rate
3. Log I/O
4. Internal locking

Why your CPU figures might differ from these

Running similar measurements on other MVS systems should give results similar to those documented in this report. However, there are several reasons why your results might differ from the results in this report. Some if these are listed below:

- The measurements were taken on a dedicated MVS system with no other applications running. Most production MVS systems have multiple concurrent applications competing for resources.
- There was sufficient real storage to eliminate paging. In a constrained system there might be additional costs due to paging.
- The MQ system had been tuned to ensure that buffer pools did not fill up and that checkpoints did not occur too frequently.
- The measurements in this report were taken with global trace off, and the accounting trace on. If you start the global trace, more CPU will be used, and if you stop the accounting trace, less CPU will be used.
- The number of batch servers can affect the amount of CPU used by the servers.
  - If you have too many servers, the CPU usage might increase because:
    - The regions get swapped out which requires CPU, see "Additional CPU used by batch servers" on page 125
    - The cost of an MQGET when a message is not immediately available is almost twice the cost of an MQGET when a message is available because the adapter re-issues the get request. This is illustrated by the figures for 1 000-byte nonpersistent messages.
  - If you do not have enough servers, the transaction response time increases.
Resources used when running the test scenario

This section describes the resources used when this application was run with messages of different size and persistence. The costs cover the CICS application putting the message, the server processing it, sending the reply back, and the CICS application getting the reply. In a real application there are additional costs due to the business logic.

The test was run for messages of size 1,000 bytes, 10,000 bytes, and 100,000 bytes, using both persistent and nonpersistent messages. For each set of results, the following information is given:

- The maximum number of transactions achieved per hour
- The amount of CPU used
- The number of MULC service units used

Notes:
1. For a given message size and persistence the CPU cost per transaction was approximately constant across the range of transaction rates measured.
2. Repeated measurements gave figures within about 10%.
3. The MULC information was extracted from the MULC report. For more information about MULC, see the MVS/ESA Support for Measured License Charges manual (GC28-1098).
4. The measurements were done on a 9672-RX5, which for MULC purposes has 1908.8523 Service Units per CPU second.
### 1000-byte persistent messages

*Maximum* transaction rate in transactions/hour | 1080000
---|---
Maximum message rate in messages/hour | 2160000

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time used by 3 CICS regions</td>
<td>94 minutes</td>
</tr>
<tr>
<td>Total CPU time used by batch server programs (TCB, SRB, and RCT)</td>
<td>143 minutes</td>
</tr>
<tr>
<td>Total CPU time used by queue manager</td>
<td>45 minutes</td>
</tr>
<tr>
<td>Total CPU time used by CICS, servers, and queue manager</td>
<td>303 minutes</td>
</tr>
<tr>
<td>Total captured CPU time</td>
<td>363 minutes</td>
</tr>
<tr>
<td>Average CPU % busy</td>
<td>73</td>
</tr>
<tr>
<td>Capture ratio</td>
<td>88%</td>
</tr>
<tr>
<td>CPU used by batch jobs, TCB, and SRB, but excluding RCT (see “Additional CPU used by batch servers” on page 125)</td>
<td>66 minutes</td>
</tr>
</tbody>
</table>

Table: CPU used to process 2 160 000 persistent 1 000-byte messages in 1 hour

<table>
<thead>
<tr>
<th>Resource</th>
<th>MULC service units per hour (millions)</th>
<th>Minutes of CPU per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 CICS regions</td>
<td>5.17</td>
<td>45</td>
</tr>
<tr>
<td>Batch servers</td>
<td>6.80</td>
<td>59</td>
</tr>
<tr>
<td>Queue manager</td>
<td>5.17</td>
<td>45</td>
</tr>
<tr>
<td>Total</td>
<td>17.33</td>
<td>149</td>
</tr>
</tbody>
</table>

Table: MQSeries MULC service units used to process 2 160 000 persistent 1 000-byte messages
1000-byte nonpersistent messages (with six server programs per server queue)

- Maximum transaction rate in transactions/hour: 1,013,760
- Maximum message rate in messages/hour: 2,027,520

Table: CPU used to process 2,027,520 nonpersistent 1,000-byte messages in 1 hour

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time used by 3 CICS regions</td>
<td>86 minutes</td>
</tr>
<tr>
<td>Total CPU time used by Batch server programs (TCB, SRB, and RCT)</td>
<td>263 minutes</td>
</tr>
<tr>
<td>Total CPU time used by queue manager</td>
<td>11 minutes</td>
</tr>
<tr>
<td>Total CPU time used by CICS, servers, and queue manager</td>
<td>360 minutes</td>
</tr>
<tr>
<td>Total time captured CPU time</td>
<td>450 minutes</td>
</tr>
<tr>
<td>Average CPU % busy</td>
<td>83</td>
</tr>
<tr>
<td>Capture ratio</td>
<td>90%</td>
</tr>
<tr>
<td>CPU used by batch jobs, TCB, and SRB, but excluding RCT (see “Additional CPU used by batch servers “on page 125)</td>
<td>55 minutes</td>
</tr>
</tbody>
</table>

Table: MQSeries MULC service units used to process 2,027,520 nonpersistent 1,000-byte messages

<table>
<thead>
<tr>
<th>Resource</th>
<th>MULC service units per hour (millions)</th>
<th>Minutes of CPU per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 CICS regions</td>
<td>4.83</td>
<td>42</td>
</tr>
<tr>
<td>Batch servers</td>
<td>7.16</td>
<td>63</td>
</tr>
<tr>
<td>Queue manager</td>
<td>1.26</td>
<td>11</td>
</tr>
<tr>
<td>Total</td>
<td>13.25</td>
<td>116</td>
</tr>
</tbody>
</table>
1000-byte nonpersistent messages (with one server program per server queue)

| Maximum transaction rate in transactions/hour | 1 205 000 |
| Maximum message rate in messages/hour | 2 405 000 |

Table: CPU used to process 2 027 520 1 000-byte nonpersistent messages in 1 hour

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time used by 3 CICS regions</td>
<td>98 minutes</td>
</tr>
<tr>
<td>Total CPU time used by Batch server programs (TCB, SRB, and RCT)</td>
<td>43 minutes</td>
</tr>
<tr>
<td>Total CPU time used by queue manager</td>
<td>12 minutes</td>
</tr>
<tr>
<td>Total CPU time used by CICS, servers, and queue manager</td>
<td>153 minutes</td>
</tr>
<tr>
<td>Total time captured CPU time</td>
<td>196 minutes</td>
</tr>
<tr>
<td>Average CPU % busy</td>
<td>37</td>
</tr>
<tr>
<td>Capture ratio</td>
<td>88%</td>
</tr>
<tr>
<td>CPU used by batch jobs, TCB, and SRB, but excluding RCT (see “Additional CPU used by batch servers” on page 125)</td>
<td>16 minutes</td>
</tr>
</tbody>
</table>

Table: MQSeries MULC used to process 2 410 000, 1 000 byte nonpersistent messages

<table>
<thead>
<tr>
<th>Resource</th>
<th>MULC service units per hour (millions)</th>
<th>Minutes of CPU per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 CICS regions</td>
<td>4.98</td>
<td>43</td>
</tr>
<tr>
<td>Batch servers</td>
<td>2.17</td>
<td>19</td>
</tr>
<tr>
<td>Queue manager</td>
<td>1.38</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>8.53</td>
<td>74</td>
</tr>
</tbody>
</table>

For 1 000-byte messages using one server application per server queue, there is a major reduction in CPU compared to using six server applications per queue. The server program was active more of the time, and had messages to process:

- It was swapped out by MVS less often (see “Additional CPU used by batch servers” on page 125)
- The application does not have to wait so often. If an application has to wait for a message, the MQGET is re-issued by the adapter when the message arrives. This leads to almost double the cost compared to when a message is available immediately. Multiple applications getting messages concurrently can lead to multiple messages being checked to see if they are available. With only one application, only the next message is checked.
10 000-byte persistent messages

| Maximum transaction rate in transactions/hour | 347 000 |
| Maximum message rate in messages/hour | 694 000 |

Table: CPU used to process 694 000 persistent 10 000-byte messages in 1 hour

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time used by 3 CICS regions</td>
<td>41 minutes</td>
</tr>
<tr>
<td>Total CPU time used by Batch server programs (TCB, SRB, and RCT)</td>
<td>40 minutes</td>
</tr>
<tr>
<td>Total CPU time used by queue manager</td>
<td>20 minutes</td>
</tr>
<tr>
<td>Total CPU time used by CICS, servers, and queue manager</td>
<td>101 minutes</td>
</tr>
<tr>
<td>Total time captured CPU time</td>
<td>116 minutes</td>
</tr>
<tr>
<td>Average CPU % busy</td>
<td>24</td>
</tr>
<tr>
<td>Capture ratio</td>
<td>81%</td>
</tr>
<tr>
<td>CPU used by batch jobs, TCB, and SRB, but excluding RCT (see “Additional CPU used by batch servers” on page 125)</td>
<td>30 minutes</td>
</tr>
</tbody>
</table>

Table: MQSeries MULC service units used to process 694 000 persistent 10 000-byte messages

<table>
<thead>
<tr>
<th>Resource</th>
<th>MULC service units per hour (millions)</th>
<th>Minutes of CPU per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 CICS regions</td>
<td>2.85</td>
<td>25</td>
</tr>
<tr>
<td>Batch servers</td>
<td>3.26</td>
<td>28</td>
</tr>
<tr>
<td>Queue manager</td>
<td>2.20</td>
<td>19</td>
</tr>
<tr>
<td>Total</td>
<td>8.31</td>
<td>72</td>
</tr>
</tbody>
</table>
### 10 000-byte nonpersistent messages

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum transaction rate in transactions/hour</td>
<td>756000</td>
</tr>
<tr>
<td>Maximum message rate in messages/hour</td>
<td>1512000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time used by 3 CICS regions</td>
<td>83 minutes</td>
</tr>
<tr>
<td>Total CPU time used by batch server programs (TCB, SRB, and RCT)</td>
<td>159 minutes</td>
</tr>
<tr>
<td>Total CPU time used by queue manager</td>
<td>12 minutes</td>
</tr>
<tr>
<td>Total CPU time used by CICS, servers, and queue manager</td>
<td>254 minutes</td>
</tr>
<tr>
<td>Total time captured CPU time</td>
<td>311 minutes</td>
</tr>
<tr>
<td>Average CPU % busy</td>
<td>60%</td>
</tr>
<tr>
<td>Capture ratio</td>
<td>86%</td>
</tr>
<tr>
<td>CPU used by batch jobs, TCB, and SRB, but excluding RCT (see “Additional CPU used by batch servers” on page 125)</td>
<td>49 minutes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>MULC service units per hour (millions)</th>
<th>Minutes of CPU per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 CICS regions</td>
<td>4.57</td>
<td>40</td>
</tr>
<tr>
<td>Batch servers</td>
<td>5.29</td>
<td>46</td>
</tr>
<tr>
<td>Queue manager</td>
<td>1.26</td>
<td>13</td>
</tr>
<tr>
<td>Total</td>
<td>14.57</td>
<td>127</td>
</tr>
</tbody>
</table>
100 000-byte persistent messages

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum transaction rate in transactions/hour</td>
<td>38040</td>
</tr>
<tr>
<td>Maximum message rate in messages/hour</td>
<td>76080</td>
</tr>
</tbody>
</table>

Table: CPU used to process 76 080 persistent 100 000-byte messages in 1 hour

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time used by 3 CICS regions</td>
<td>8 minutes</td>
</tr>
<tr>
<td>Total CPU time used by batch server programs (TCB, SRB, and RCT)</td>
<td>8 minutes</td>
</tr>
<tr>
<td>Total CPU time used by queue manager</td>
<td>6 minutes</td>
</tr>
<tr>
<td>Total CPU time used by CICS, servers, and queue manager</td>
<td>22 minutes</td>
</tr>
<tr>
<td>Total time captured CPU time</td>
<td>24 minutes</td>
</tr>
<tr>
<td>Average CPU % busy</td>
<td>6</td>
</tr>
<tr>
<td>Capture ratio</td>
<td>66%</td>
</tr>
<tr>
<td>CPU used by batch jobs, TCB, and SRB, but excluding RCT (see “Additional CPU used by batch servers” on page 125)</td>
<td>7 minutes</td>
</tr>
</tbody>
</table>

Table: MQSeries MULC service units used to process 76 080 persistent 100 000-byte messages

<table>
<thead>
<tr>
<th>Resource</th>
<th>MULC service units per hour (millions)</th>
<th>Minutes of CPU per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 CICS regions</td>
<td>0.69</td>
<td>6</td>
</tr>
<tr>
<td>Batch servers</td>
<td>0.73</td>
<td>6</td>
</tr>
<tr>
<td>Queue manager</td>
<td>0.68</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>2.105</td>
<td>18</td>
</tr>
</tbody>
</table>
### 100 000-byte nonpersistent messages

| Maximum transaction rate in transactions/hour | 341000 |
| Maximum message rate in messages/hour         | 682000 |

#### Table: CPU used to process 682 000 nonpersistent 100 000-byte messages in 1 hour

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time used by 3 CICS regions</td>
<td>86 minutes</td>
</tr>
<tr>
<td>Total CPU time used by batch server programs (TCB, SRB, and RCT)</td>
<td>84 minutes</td>
</tr>
<tr>
<td>Total CPU time used by queue manager</td>
<td>23 minutes</td>
</tr>
<tr>
<td>Total CPU time used by CICS, servers, and queue manager</td>
<td>193 minutes</td>
</tr>
<tr>
<td>Total time captured CPU time</td>
<td>207 minutes</td>
</tr>
<tr>
<td>Average CPU % busy</td>
<td>48</td>
</tr>
<tr>
<td>Capture ratio</td>
<td>72%</td>
</tr>
<tr>
<td>CPU used by batch jobs, TCB, and SRB, but excluding RCT (see “Additional CPU used by batch servers” on pag125)</td>
<td>74 minutes</td>
</tr>
</tbody>
</table>

#### Table: MQSeries MULC service units used to process 682 000 nonpersistent 100 000-byte messages

<table>
<thead>
<tr>
<th>Resource</th>
<th>MULC service units per hour (millions)</th>
<th>Minutes of CPU per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 CICS regions</td>
<td>7.09</td>
<td>62</td>
</tr>
<tr>
<td>Batch servers</td>
<td>7.54</td>
<td>66</td>
</tr>
<tr>
<td>Queue manager</td>
<td>2.66</td>
<td>23</td>
</tr>
<tr>
<td>Total</td>
<td>17.29</td>
<td>150</td>
</tr>
</tbody>
</table>
Adjusting for uncollected CPU time

The total CPU used, as recorded by MVS, does not match the figure reported by the hardware. This is because of the overhead of collecting the data, and other events (for example, I/O and other interrupts).

One way to compensate for this is to allocate the unrecorded CPU to regions in proportion to the amount of CPU used by that region. This can be calculated by dividing the CPU used by the "capture ratio", where the capture ratio is defined as the percentage of the total CPU recorded against all address spaces divided by the total CPU used. In practice this is calculated from the CPU reported in the ALL ALL ALL line of the RMF report divided by (time period * number of engines * average % busy of the processors).

The sum of the adjusted amount of CPU used will equal the amount of CPU reported by the hardware, and the adjusted values for a region will give a more accurate value for capacity planning purposes. The figures in the tables are not adjusted for capture ratio.

Additional CPU used by batch servers

For the batch servers, the RMF reports show that there were many swaps per second and a lot of CPU was spent in the Region Control Task (RCT). This RCT CPU is used to swap the address space in and out. When a server issues an MQGET with wait and there is no message available in a short time, the MVS SRM might swap out the address space if there is a resource shortage. See also “Avoiding significant swapping of batch MQ applications “ on page 20.

You can reduce this CPU time by reducing the number of batch servers so that there are always messages for the batch server to process and they do not have to wait for messages. However, this will affect the transaction response time.

In the tables showing CPU usage for messages of different size and persistence, the cost of the batch servers, including the RCT time, has been included in the total cost. The cost of the batch servers, excluding the RCT time, has also been included in the tables to show how much CPU was used doing useful work.
What happens as the transaction rate increases

This section discusses how the transaction rate affects the performance of your system. We used the test scenario with 1000-byte persistent messages. TPNS was used to drive the transactions at different rates.

There is a smooth increase as the transaction rate increases—there is no sign of an abrupt change, which would indicate that the maximum transaction rate has been achieved.

- There is a linear increase in CPU % busy with increasing transaction rate.
- At a high transaction rate, the average time for the transaction is 260 milliseconds with a standard deviation of 177. At a low transaction rate, the response time is about 40 milliseconds.

  - The increased response time at higher transaction rate is caused by a combination of the following factors:
    - I/Os to the log data set take longer
    - The transactions have to wait for an MQ TCB in the CICS address space

- Having a greater number of CICS regions will help relieve the latter problem.

Log data set characteristics

Each MQ log was placed on its own RVA2-T82 DASD volume, and the DASD activity reports in RMF show the I/O activity associated with the active log data sets. In this case the average DASD response time increases from 3 milliseconds to 6 milliseconds and the device utilization increases from about 50% to 80% busy.

The variation of I/O rate, DASD response time, and % device utilized are discussed below.

The EXEC CICS SYNCPOINT command issues the commit to the queue manager. The queue manager has to ensure that all of the recoverable data relating to the transaction has been written out to disk, causing a write operation to the log data set if the data has not been written out previously.

At a very low transaction rate, each EXEC CICS SYNCPOINT causes one write operation to the log data set, and as little as 1 page might be written per log I/O.

As the transaction rate increases, several transaction might be committing concurrently. In this case several log buffers might be written to disk in one I/O. This reduces the number of I/Os required, but the I/Os take slightly longer because more data has to be transferred. This causes the device response time and the % utilization of the disk to increase.

At a high transaction rate, many pages (possibly up to 15) are being written in each I/O. This is explained in “APPENDIX B. The log manager” on page 181.
Capacity planning for a CICS requester / Batch reply model with remote messages

This chapter describes the costs of requester/reply applications using remote queuing. The messages are moved between queue managers in different MVS images in the same sysplex, using TCP/IP.

Many customers who use distributed queuing have distributed queue managers sending individual requests to a server queue manager on MVS that processes the request and sends a response back. Typically the messages are around 1000-bytes or less.

We managed to achieve 120 message pairs per second for persistent messages, and 220 messages pairs per second for nonpersistent messages. This is the worst case scenario, with more messages per batch you can achieve significantly better throughput.

The approximate CPU time to process this request for persistent messages is 20 CPU milliseconds, and 15 CPU milliseconds for nonpersistent messages. If we allow 10 CPU milliseconds as a typical application cost, the total CPU needed to process a request message and send a response back is of the order of 30 ms for persistent, and 25 ms for nonpersistent messages.

The measurements were taken on a 9672-R55 with 5 engines and 2472.1 service units per second.

**Measurement scenario**

![Diagram showing the measurement scenario](Figure: Set up for the capacity planning measurements)

- TPNS was used to execute transactions in three CICS regions. (Three CICS regions were used because in some cases we got reduced throughput when using only one CICS region.)
- The transactions put a number of messages to a remote queue and committed the request. The transaction issued an **MQGET** with wait on its local reply-to queue, for the same number of messages. This number of messages then becomes the *achieved batch size* on each channel.
- Each remote queue went to a unique transmission queue, with one channel processing each transmission queue.
- At the remote end, the messages were put to a unique remote queue, with a unique transmission queue per remote queue and one channel per transmission queue. This resulted in a loop back with no application processing.
  If there had been a server application processing messages, the reply messages would be likely to be sent back in batches of 1 regardless of the number of messages put in one unit of work in the
CICS application. In a different scenario, where the round trip time is not so important, specifying a BATCHINT value would have given the expected number of messages per batch.

- Each of the three CICS systems had 100 simulated terminals, so there were 300 channels each way between the two queue managers.

Notes on the scenario

- The originating end is called system A and the remote end is called system B.
- The channels were defined with a batch size of 50, but for many of the measurements the achieved batch size was less (as is typical for requester/reply applications).
- The CICS transaction put a different number of messages per measurement so that the effects of different numbers of messages in a batch could be determined. For persistent messages, the messages flowed as one batch because they were all processed within syncpoint. However, the channel initiator processes nonpersistent fast message out of syncpoint, so the desired batch size was not always obtained.
- A "message pair" comprises the request message sent to the remote end and the returned reply. If the CICS transaction puts 8 messages at a time, and receives 8 replies, this is regarded as 8 message pairs.
- The work done by system A is typical of the work done by a server back end, though the calls are executed in a different order. For example, on a server MVS system, a message would come in on a channel, be retrieved by a CICS transaction, a reply message put by CICS, and the reply message sent down a channel. In the measured scenario the CICS transaction put a message, this was sent down a channel, the reply message came in on a channel and the CICS application got it.
- System B does not have any application processing and so has fewer MQI calls than a true server.
- The measurements were taken using MQSeries for MVS/ESA Version 1.2, OS/390 Version 2.4, and TCP/IP Version 3.2 (IUCV)

```
* Define the remote queue used for putting messages
DEFINE QR(RCP0000) XMITQ(XCP0000) RNAME(RCP0000) RQMNAME(VICB)
* Define the transmission queue
DEFINE QL(XCP0000) USAGE(XMITQ) STGCLASS(P1)
* Define the sending channel to queue manager VICB
DEFINE CHANNEL(CSCP0000 ) +
   CHLTYPE(SDR) TRPTYPE(TCP) CONNAME('xx.xx.xx.xx(pppp)') +
   XMITQ(XCP0000 )
* Define the channel from queue manager VICB
DEFINE CHANNEL(CRCP0000 ) CHLTYPE(RCVR) TRPTYPE(TCP )
* Define the local reply-to queue
DEFINE QL(CP0000) STGCLASS(P1)
```

Figure: Sample MQSeries definitions for system A For a TPNS terminal with LU name CP0000, the messages were put to queue RCP0000 and the MQGET issued to queue CP0000
* Define the channel from VICA
DEFINE CHANNEL(CSCP0000) CHLTYPE(RCVR) TRPTYPE(TCP)

* Define the remote queue so message get sent straight back to VICA
DEFINE QR(RCP0000) XMITQ(XCP0000) RNAME(CP0000) RQMNAME(VICA)

* Define the transmission queue
DEFINE QL(XCP0000) USAGE(XMITQ) STGCLASS(P1)

* Define the channel to queue manager VICA
DEFINE CHANNEL(CRCP0000) +
  CHLTYPE(SDR) TRPTYPE(TCP) CONNAME('xx.xx.yy.yy(pppp)') +
  XMITQ(XCP0000)

Figure: Sample MQSeries definitions for system B

### Number of adapters and dispatchers

Within the channel initiator address space, adapter TCBs process MQ requests and dispatcher TCBs process the communications with TCP/IP and APPC. The number of these adapter and dispatcher TCBs can be specified using the channel initiator parameter module (CSQXPARM). The supplied sample uses a ratio of 8 adapters to 5 dispatchers and the MQSeries for MVS/ESA System Management Guide suggests 1 dispatcher for each 50 channels.

The ratio of 1 dispatcher for each 50 channels is not critical, but having a large number of channels per dispatcher can effect the throughput and CPU used by the channel initiator.

For example, using 300 sender channels and 300 receiver channels (600 in total):

- With 8 adapters and 5 dispatchers the rate was 115 message pairs per second.
- With 20 adapters and 12 dispatchers the rate was 155 message pairs per second, but the average cost per message pair increased by 10%.

The increased throughput is due to less queuing for a TCB, and the increased cost is due to the TCB being suspended and resumed more frequently.

Alterning the ratio of adapters to dispatchers had little impact on the measured throughput rate. If you store messages for deferred processing (overnight for example) the messages are likely to be on page sets rather than being held in memory. In this case, if you have many page sets, you might get improved throughput by specifying more adapters to allow more I/O in parallel.

### TCP/IP tuning parameters

During testing, TCP/IP sometimes produced message about a shortage of a resource. The TCP/IP definitions were increased and the measurements repeated. The figure below shows the final values as shown by the NETSTAT command.
<table>
<thead>
<tr>
<th>TCP/IP parms name</th>
<th>Netstat name</th>
<th># alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACBPOOLSIZE</td>
<td>ACB</td>
<td>14000</td>
</tr>
<tr>
<td>CCBPOOLSIZE</td>
<td>CCB</td>
<td>2500</td>
</tr>
<tr>
<td>DATABUFFERPOOLSIZE</td>
<td>Dat buf</td>
<td>2400</td>
</tr>
<tr>
<td>SMALLDATABUFFERPOOLSIZE</td>
<td>Sm dat buf</td>
<td>4800</td>
</tr>
<tr>
<td>TINYDATABUFFERPOOLSIZE</td>
<td>Tiny dat buf</td>
<td>500</td>
</tr>
<tr>
<td>ENVELOPEPOOLSIZE</td>
<td>Env</td>
<td>750</td>
</tr>
<tr>
<td>LARGEENVELOPEPOOLSIZE</td>
<td>Lrg env</td>
<td>50</td>
</tr>
<tr>
<td>RCBPOOLSIZE</td>
<td>RCB</td>
<td>100</td>
</tr>
<tr>
<td>SCBPOOLSIZE</td>
<td>SCB</td>
<td>768</td>
</tr>
<tr>
<td>SKCBPOOLSIZE</td>
<td>SKCB</td>
<td>1600</td>
</tr>
<tr>
<td>TCBPOOLSIZE</td>
<td>TCB</td>
<td>9000</td>
</tr>
<tr>
<td>UCBPOOLSIZE</td>
<td>UCB</td>
<td>100</td>
</tr>
<tr>
<td>ADDRESSTRANSITIONPOOLSIZE</td>
<td>Add Xlate</td>
<td>1500</td>
</tr>
<tr>
<td>IPROUTEPOOLSIZE</td>
<td>IP Route</td>
<td>600</td>
</tr>
</tbody>
</table>
Summary of measurements

These measurements were taken on a 9672-R55 processor with 5 engines and 2472.1 service units per second.

<table>
<thead>
<tr>
<th>Message size in bytes</th>
<th>Persistence</th>
<th>Batch size</th>
<th>Msg-pairs per second</th>
<th>Msg-pairs per hour</th>
<th>CPU milliseconds used per message pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000</td>
<td>Persistent</td>
<td>1</td>
<td>120</td>
<td>432 000</td>
<td>21.0</td>
</tr>
<tr>
<td>1 000</td>
<td>Persistent</td>
<td>2</td>
<td>200</td>
<td>720 000</td>
<td>12.8</td>
</tr>
<tr>
<td>1 000</td>
<td>Persistent</td>
<td>50</td>
<td>470</td>
<td>1 710 000</td>
<td>4.5</td>
</tr>
<tr>
<td>1 000</td>
<td>Nonpersistent</td>
<td>1</td>
<td>220</td>
<td>792 000</td>
<td>14.1</td>
</tr>
<tr>
<td>1 000</td>
<td>Nonpersistent</td>
<td>2</td>
<td>265</td>
<td>954 000</td>
<td>10.1</td>
</tr>
<tr>
<td>1 000</td>
<td>Nonpersistent</td>
<td>50</td>
<td>625</td>
<td>2 250 000</td>
<td>5.8</td>
</tr>
<tr>
<td>10 000</td>
<td>Persistent</td>
<td>1</td>
<td>60</td>
<td>216 000</td>
<td>26.2</td>
</tr>
<tr>
<td>10 000</td>
<td>Persistent</td>
<td>2</td>
<td>72</td>
<td>259 200</td>
<td>17.1</td>
</tr>
<tr>
<td>10 000</td>
<td>Persistent</td>
<td>8</td>
<td>85</td>
<td>306 000</td>
<td>11.6</td>
</tr>
<tr>
<td>10 000</td>
<td>Nonpersistent</td>
<td>1</td>
<td>200</td>
<td>720 000</td>
<td>15.2</td>
</tr>
<tr>
<td>10 000</td>
<td>Nonpersistent</td>
<td>2</td>
<td>212</td>
<td>763 200</td>
<td>15.8</td>
</tr>
<tr>
<td>10 000</td>
<td>Nonpersistent</td>
<td>8</td>
<td>205</td>
<td>738 000</td>
<td>15.7</td>
</tr>
<tr>
<td>100 000</td>
<td>Persistent</td>
<td>1</td>
<td>6.9</td>
<td>24 840</td>
<td>64.6</td>
</tr>
<tr>
<td>100 000</td>
<td>Nonpersistent</td>
<td>1</td>
<td>62</td>
<td>223 200</td>
<td>38.7</td>
</tr>
</tbody>
</table>

Note:

1. The CPU milliseconds user per message pair is the cost on system A and includes the CPU from CICS, channel initiator, queue manager, and TCP/IP.

2. The benefits of batching can easily be seen, for example with 1 000-byte persistent messages, going from batch size 1 to batch size 2 gives 60% increase in throughput (432 000 to 720 000) for almost the same CPU cost (21.0 and 2*12.8).

3. For a particular scenario the costs are approximately linear with throughput, so half the throughput will use approximately half the CPU. However in practice, with the same number of channels, an increase in throughput might result in little increase in CPU overall because there might be more messages per batch.

4. These figures were obtained from 20 minutes of measurement and scaled to an hour. Within the measurement period, the throughput varied by around 10% as events like the end-of-log processing and checkpointing caused a temporary reduction in throughput.

5. The CPU milliseconds user per message pair includes the cost of a minimal CICS application processing messages. A truer cost per message pair will include application processing, which might of the order of 10 CPU milliseconds per message pair.
Effect of batch size on throughput

The BATCHSZ parameter of a channel is the maximum number of messages that will be sent in a batch. In practice the number of messages in a batch (the achieved batch size) might be smaller than the specified value because the batch ends when BATCHSZ messages are sent, or there are no more messages on the transmission queue. If there was only one message on the transmission queue, this would be the only message in the batch, giving an achieved batch size of 1.

Using BATCHINT to keep a batch open for longer can increase the number of messages per batch but, for persistent messages this will add a delay before the message are available at the remote end.

In transactional type requests, where individual messages are coming in from a remote queue manager, there are usually only a few messages, possibly only one, per batch.

Generally, as the number of messages in a batch increases you get increased throughput up to a peak value, then the throughput gradually decreases as the number of messages per batch is further increased. This is due to two effects:

1. Less end-of-batch processing.
   As the number of messages per batch increases, end-of-batch activities like commits are performed less frequently so more time is available for sending messages.

2. Higher MQ costs.
   As the volume of data flowing into and out of the queue manager increases and there are more messages per unit of work (messages per batch), applications compete for resources within the queue manager and the elapsed time and the CPU cost for MQI calls increases.

The value of the batch size at peak throughput depends on the message size in addition to the amount of activity in the queue manager; it is difficult to say what the best value is. However for 1 000-byte messages the default BATCHSZ of 50 generally gives good throughput; for 100 000-byte messages the optimum batch size is less than 10.

The BATCHINT parameter on a channel is the minimum time that a channel will keep a batch open. Messages arriving within the BATCHINT interval will be sent as one batch. For example, if the BATCHINT is set to 10 milliseconds, and the second message arrives 8 milliseconds after the first, both messages will be in the same batch. This can reduce the CPU usage, but messages can take longer to be sent because there is the BATCHINT delay from the time the first message is sent.

If the achieved batch size is nearly always 1, setting BATCHSZ to 1 can save more than 5% CPU. This is because the end-of-batch flag is sent with the message, and does not require a separate flow.

Reduced MQGET costs when messages available immediately

When an application issues an MQGET with wait request and a message is not available, control is returned to the adapter and the task is suspended. When a message becomes available, the task is resumed and the adapter reissues the MQI request for the application. In effect, two MQGET request have been issued.

When the message is available immediately the task does not have to wait and so the CPU cost is almost half the cost compared to when the application has to wait.
## Breakdown of costs

### Resources used processing 1 000-byte messages

<table>
<thead>
<tr>
<th>Persistence</th>
<th>P</th>
<th>N</th>
<th>P</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of message pairs per transaction</td>
<td>1</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Transactions per second</td>
<td>120</td>
<td>220</td>
<td>9.4</td>
<td>12.5</td>
</tr>
<tr>
<td>Round trip message pairs per second</td>
<td>120</td>
<td>220</td>
<td>475</td>
<td>625</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU millisecs time used per message pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Queue manager</td>
</tr>
<tr>
<td>A: Channel initiator</td>
</tr>
<tr>
<td>A: TCP/IP</td>
</tr>
<tr>
<td>A: CICS</td>
</tr>
<tr>
<td>A: Sum of CPU time</td>
</tr>
<tr>
<td>B: Queue manager</td>
</tr>
<tr>
<td>B: Channel initiator</td>
</tr>
<tr>
<td>TCP/IP</td>
</tr>
<tr>
<td>B: Sum of CPU time</td>
</tr>
</tbody>
</table>

| Transactions per hour | 432 000 | 792 000 | 34 200 | 45 000 |
| Round trip message pairs per hour | 432 000 | 792 000 | 1 710 000 | 2 250 000 |

<table>
<thead>
<tr>
<th>Total CPU minutes used by CICS, queue manager, channel initiator and TCP/IP in an hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
</tr>
<tr>
<td>B:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total CPU used in MVS systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Total CPU time captured minutes/hour</td>
</tr>
<tr>
<td>A: Average CPU % busy</td>
</tr>
<tr>
<td>A: Capture ratio</td>
</tr>
<tr>
<td>B: Total CPU time captured minutes/hour</td>
</tr>
<tr>
<td>B: Average CPU % busy</td>
</tr>
<tr>
<td>B: Capture ratio</td>
</tr>
</tbody>
</table>

1. A larger batch size gives better throughput.
2. For persistent messages, the limit to throughput is the speed at which data can be written to the log data sets. Nonpersistent messages are limited by the amount of CPU available.
3. TCP/IP costs:
a. The TCP/IP costs are approximately the same for 1,000-byte messages with a batch size of 1, because the flows are the same for persistent and nonpersistent messages.

b. For larger batch sizes there are fewer end-of-batch flows so there is less traffic flowing over TCP/IP.

c. For a batch size of 50, there is a small increase in cost for nonpersistent messages compared to persistent messages. This is the same effect as described in "Reduced MQGET costs when messages available immediately" on page 132 and is caused by the returning nonpersistent messages being put out of syncpoint at the remote end. It can happen that the return channel has sent some messages, but the next message is not yet available, so an end-of-batch flow is sent. This results in an achieved batch size between 1 and 50, whereas with persistent messages, all the messages are sent in one batch because they were all put within syncpoint.

4. Cost within the queue manager:
   a. The costs within the queue manager for persistent messages are higher than nonpersistent messages. This is because most of the commit activity is performed on a task within the queue manager, and the CPU for this is charged to the queue manager. The increased cost covers items like writing to the log data sets.
   b. The commit cost per message is smaller when there are many messages per unit of work because the overhead of the commits is spread over a larger number of messages.
   c. In the queue manager there are background tasks that remove empty pages from queues, and make these pages available again. In system A there are twice as many messages processed as in system B (because there is no application in system B) so the cost of the background tasks is higher.

5. Costs in the CICS regions:
   a. For batch size 1, the CICS costs show that the CPU used for persistent messages is slightly higher than for nonpersistent messages. This is because of the additional work that needs to be done to make the messages recoverable.
   b. Although the same applies for batch size 50, the costs of the nonpersistent messages are higher due to the effect described in "Reduced MQGET costs when messages available immediately" on page 132.

6. Costs in the channel initiator:
   a. As with the CICS regions, the CPU used for persistent messages is slightly higher than for nonpersistent messages. This is because of the additional work that needs to be done to make the messages recoverable, both in the queue manager and in the channel initiator.
   b. For a batch size of 1, the channel initiators at both ends are processing the same number of messages and batches. The increased cost in system A is due to the slightly higher costs as the channel initiator is competing for resources with the CICS regions in the queue manager.
Resources used processing 10 000-byte messages

<table>
<thead>
<tr>
<th>Persistence</th>
<th>P</th>
<th>N</th>
<th>P</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of message pairs per transaction (batch size)</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Transactions per second</td>
<td>60</td>
<td>200</td>
<td>10.6</td>
<td>25.6</td>
</tr>
<tr>
<td>Round trip message pairs per second</td>
<td>60</td>
<td>200</td>
<td>85</td>
<td>205</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU time used per message pair in CPU milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Queue manager</td>
</tr>
<tr>
<td>A: Channel initiator</td>
</tr>
<tr>
<td>A: TCP/IP</td>
</tr>
<tr>
<td>A: CICS</td>
</tr>
<tr>
<td>A: Sum of CPU time</td>
</tr>
<tr>
<td>B: Queue manager</td>
</tr>
<tr>
<td>B: Channel initiator</td>
</tr>
<tr>
<td>B: TCP/IP</td>
</tr>
<tr>
<td>B: Sum of CPU time</td>
</tr>
</tbody>
</table>

| Transactions per hour | 216 000 | 720 000 | 38 160 | 92 160 |
| Round trip message pairs per hour | 216 000 | 720 000 | 306 000 | 738 1000 |

<table>
<thead>
<tr>
<th>Total CPU minutes used by CICS, queue manager, channel initiator and TCP/IP in an hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
</tr>
<tr>
<td>B:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total CPU used in MVS systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Total CPU time captured minutes/hour</td>
</tr>
<tr>
<td>A: Average CPU % busy</td>
</tr>
<tr>
<td>A: Capture ratio</td>
</tr>
<tr>
<td>B: Total CPU time captured minutes/hour</td>
</tr>
<tr>
<td>B: Average CPU % busy</td>
</tr>
<tr>
<td>B: Capture ratio</td>
</tr>
</tbody>
</table>
Resources used processing 100 000-byte messages

Table: CPU used to process 100 000-byte messages

<table>
<thead>
<tr>
<th>Persistence</th>
<th>P</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of message pairs per transaction (Batch size)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Transactions per second</td>
<td>6.9</td>
<td>62</td>
</tr>
<tr>
<td>Round trip message pairs per second</td>
<td>6.9</td>
<td>62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU time used per message pair in CPU milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: Queue manager</td>
</tr>
<tr>
<td>A: Channel initiator</td>
</tr>
<tr>
<td>A: TCP/IP</td>
</tr>
<tr>
<td>A: CICS</td>
</tr>
<tr>
<td>A: Sum of CPU time</td>
</tr>
<tr>
<td>B: Queue manager</td>
</tr>
<tr>
<td>B: Channel initiator</td>
</tr>
<tr>
<td>B: TCP/IP</td>
</tr>
<tr>
<td>B: Sum of CPU time</td>
</tr>
</tbody>
</table>

Transactions per hour

| Round trip message pairs per hour | 24 840 | 223 200 |

Total CPU minutes used by CICS, queue manager, channel initiator and TCP/IP in an hour

| A: | 26 | 144 |
| B: | 17 | 92 |

Total CPU used in MVS systems

| A: Total CPU time captured minutes/hour | 29 | 146 |
| A: Average CPU % busy | 11 | 53 |
| A: Capture ratio | 86 | 91 |
| B: Total CPU time captured minutes/hour | 19 | 99 |
| B: Average CPU % busy | 7 | 36 |
| B: Capture ratio | 91 | 92 |

Note: For persistent messages, the limit to throughput is the speed at which data can be logged.
Performance implications of very large messages

Introduction

This chapter contains the results of very large message measurements for MQSeries for OS/390 Version 2.1 with APAR PQ33000 Very Large Message Support PTFs applied.

General performance implications of very large messages

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

- Page set I/O is more likely with very large messages than with the same amount of data in smaller messages.
- Very large messages will result in increased virtual storage usage in applications and in the MQSeries channel initiator. This is likely to cause increased real storage usage in applications and MQSeries buffer pools. These considerations could cause an increase in response times compared to using the same amount of data in several smaller messages.

The maximum number of channels all transmitting 100-MB messages is unlikely to exceed 15 because of virtual storage limitations. The use of BATCHSZ(1) is recommended for any channel transmitting very large messages.

Results summary

The following test scenarios were measured for message sizes varying from 4 MB to the maximum of 100 MB, both persistent and nonpersistent. All MQPUTs and MQGETs were performed by simple batch applications with no business application logic. MQ calls were within syncpoint, with each message individually committed. MQCONN, MQOPEN, MQCLOSE and MQDISC calls were performed outside of the measurement loop, that is the CPU costs of these calls are not include in the results given below.

- Single queue manager:
  - Single application performing MQPUT followed by MQGET of each message
  - Single application; MQPUT only
  - Single application; MQGET only
  - Two applications performing MQPUTs in parallel
  - Two applications performing MQGETs in parallel
  - Application performing MQPUT followed by MQGET, with parallel application performing MQPUT/MQGET of 1000-byte messages
  - Two applications, one preforming MQPUT only, the other performing MQGET in parallel (from the same, pre-loaded queue)
  - Single application; MQPUT to a buffer pool that is too small
  - Single application; MQGET from a buffer pool that is too small

- Two queue managers on separate MVS images, TCP/IP connectivity via CTCs. Transmission queues were pre-loaded with messages prior to starting the measurements:
  - Single sender/receiver TCP/IP channel pair; messages flowing in one direction only, with a batch application removing messages at the receiving end.
  - Two sender/receiver TCP/IP channel pairs; messages flowing in one direction only, with two batch applications removing messages at the receiving end. In the single queue manager
scenarios, application and queue manager CPU usage was determined by the batch application. In the distributed queuing scenarios, RMF monitor I was used to measure CPU usage.

How do CPU costs scale with increasing message size?

Local queuing

- **Persistent messages**: In all cases, application and queue manager CPU usage for individual MQPUTs and MQGETs of persistent large messages increased linearly with increasing message size. Typically, on a 9672-R56 processor, total CPU costs for persistent messages were:
  - MQPUT: 25 ms per MB
  - MQGET: 8 ms per MB Note: both calls were in syncpoint and include the cost of the commit.

- **Nonpersistent messages**: In the tests with individual MQPUTs and MQGETs of nonpersistent messages, application CPU usage increased linearly with message size, however, queue manager CPU usage increased at a nonlinear rate. Typically, on a 9672-R56 processor, total CPU costs for nonpersistent message varied from:
  - MQPUT: 12 ms per MB
  - MQGET: 8 ms per MB with 4-MB and 10-MB messages, up to:
  - MQPUT: 24 ms per MB
  - MQGET: 15 ms per MB with 100-MB messages.

Note: All calls were in syncpoint and include the cost of the commit.

This non-linearity, which is seen in the Qmgr SRB CPU time, is due to the way the MQ soft log process frees chained control blocks during commit/backout processing.

- In tests where the batch application issued MQPUT followed by MQGET for each message, all of the results very closely matched the combined results of the individual MQPUT and MQGET tests.

Distributed queuing

- With a single channel pair and persistent messages, both sender-end and receiver-end total CPU usage showed a high degree of linearity as message size was increased from 4 MB to 100 MB. With nonpersistent messages, sender-end CPU usage increased linearly, while receiver-end CPU showed a slight non-linearity, as with local queuing, attributable to a non-linear increase in queue manager SRB time. It should be noted that the receiver-end queue manager SRB includes CPU time attributable to the batch application which removed (MQGET) messages as they arrived.

- With two channel pairs processing messages in parallel, total CPU usage was close to that of a single channel up to message sizes of 20 MB. Messages of 50 MB gave total CPU usage increases of around 10% compared with a single channel pair for both persistent and nonpersistent messages. 100-MB messages gave CPU usage increases of between 7% and 27%.
How do elapsed times scale with increasing message size?

Local queuing

- Elapsed times for individual **MQPUT**s and **MQGET**s followed the same pattern as CPU usage, increasing linearly with message size for persistent messages, and non-linearly for nonpersistent messages.

Distributed queuing

- In all of the distributed queuing tests, elapsed times increased linearly as message size was increased from 4 MB to 100 MB.

What effect do large messages have on other MQ activity?

Local queuing

- Where two large messages were processed in parallel, elapsed times and application and queue manager CPU usage all showed some increase compared with the individual message tests. **MQGET**s were least affected with total CPU increases of between 3% and 13% for nonpersistent messages; 4% and 29% for persistent messages. Persistent **MQPUT** showed total CPU increases of 2% to 16%, while nonpersistent **MQPUT**s showed the most significant increases of 17% to 62%, the latter with 100 MB messages.

- Where small, 1000-byte messages were processed in parallel with very large messages, the small message application CPU usage showed only a small increase (between 1% and 6%) as the parallel message size was increased from 4 MB to 100 MB. This was true for both persistent and nonpersistent messages. However, the elapsed time for persistent messages was significantly lengthened because it shares use of the log with the large message. Thus a single program using small persistent messages would have a significantly reduced maximum throughput.

How does a buffer pool that is too small affect large messages?

Local queuing

- Where the buffer pool size was too small to hold a single 100-MB message (approximately 26000 pages are required), very significant increases in application and queue manager CPU usage and elapsed times were observed. This was most noticeable with nonpersistent 100-MB messages, where both elapsed time and CPU cost exceeded that of persistent 100-MB messages. Buffer manager statistics show that with nonpersistent messages the "buffer pool 95% full" condition was reached, thus forcing synchronous page writes. Persistent message buffers were slowed down by logging I/O and so asynchronous page writes initiated at 85% full were able to keep the buffer pool below 95% full.
Results - graphical

Local queuing

Figure: OS/390 CPU usage. MQPUT then MQGET of persistent messages, 4-MB to 100-MB

Figure: OS/390 CPU usage. MQPUT then MQGET of nonpersistent messages, 4-MB to 100-MB
Distributed queuing

Figure: OS/390 CPU usage. Qmgr to qmgr (single direction), persistent messages, 4-MB to 100-MB

Figure: OS/390 CPU usage. Qmgr to qmgr (single direction), nonpersistent messages, 4-MB to 100-MB
# Results - tabular

## Local queuing

### Single application putting then getting, 4-MB to 100-MB messages

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Elapsed time (s)</th>
<th>9672-R56 CPU (s/msg) Application</th>
<th>Qmgr</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Nonpers</td>
<td>0.09</td>
<td>0.066</td>
<td>0.010</td>
<td>0.076</td>
</tr>
<tr>
<td>10</td>
<td>Nonpers</td>
<td>0.23</td>
<td>0.165</td>
<td>0.036</td>
<td>0.201</td>
</tr>
<tr>
<td>20</td>
<td>Nonpers</td>
<td>0.57</td>
<td>0.353</td>
<td>0.110</td>
<td>0.463</td>
</tr>
<tr>
<td>50</td>
<td>Nonpers</td>
<td>1.72</td>
<td>0.912</td>
<td>0.553</td>
<td>1.465</td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>4.49</td>
<td>1.917</td>
<td>2.031</td>
<td>3.948</td>
</tr>
<tr>
<td>4</td>
<td>Pers</td>
<td>1.53</td>
<td>0.102</td>
<td>0.026</td>
<td>0.128</td>
</tr>
<tr>
<td>10</td>
<td>Pers</td>
<td>4.37</td>
<td>0.256</td>
<td>0.063</td>
<td>0.319</td>
</tr>
<tr>
<td>20</td>
<td>Pers</td>
<td>7.37</td>
<td>0.541</td>
<td>0.119</td>
<td>0.660</td>
</tr>
<tr>
<td>50</td>
<td>Pers</td>
<td>17.85</td>
<td>1.349</td>
<td>0.297</td>
<td>1.646</td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>40.62</td>
<td>2.744</td>
<td>0.692</td>
<td>3.436</td>
</tr>
</tbody>
</table>

### Single application putting only, 4-MB to 100-MB messages

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Elapsed time (s)</th>
<th>9672-R56 CPU (s/msg) Application</th>
<th>Qmgr</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>53.83</td>
<td>8.954</td>
<td>2.066</td>
<td>11.020</td>
</tr>
<tr>
<td>4</td>
<td>Pers</td>
<td>1.49</td>
<td>0.071</td>
<td>0.024</td>
<td>0.095</td>
</tr>
<tr>
<td>10</td>
<td>Pers</td>
<td>3.79</td>
<td>0.180</td>
<td>0.060</td>
<td>0.240</td>
</tr>
<tr>
<td>20</td>
<td>Pers</td>
<td>7.68</td>
<td>0.363</td>
<td>0.118</td>
<td>0.481</td>
</tr>
<tr>
<td>50</td>
<td>Pers</td>
<td>19.01</td>
<td>0.937</td>
<td>0.285</td>
<td>1.222</td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>35.02</td>
<td>1.922</td>
<td>0.580</td>
<td>2.502</td>
</tr>
<tr>
<td>100 small BP</td>
<td>Pers</td>
<td>45.07</td>
<td>1.846</td>
<td>1.861</td>
<td>3.707</td>
</tr>
</tbody>
</table>
### Single application getting only, 4-MB to 100-MB messages

**Table:** Very large message 9672-R56 CPU usage and elapsed time

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Elapsed time (s)</th>
<th>9672-R56 CPU (s/msg)</th>
<th>Application</th>
<th>Qmgr</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Nonpers</td>
<td>0.03</td>
<td>0.026</td>
<td>0.004</td>
<td>0.030</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Nonpers</td>
<td>0.08</td>
<td>0.065</td>
<td>0.014</td>
<td>0.079</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Nonpers</td>
<td>0.23</td>
<td>0.137</td>
<td>0.044</td>
<td>0.181</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Nonpers</td>
<td>0.69</td>
<td>0.337</td>
<td>0.228</td>
<td>0.566</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>1.80</td>
<td>0.692</td>
<td>0.846</td>
<td>1.538</td>
<td></td>
</tr>
<tr>
<td>100 small BP</td>
<td>Nonpers</td>
<td>13.53</td>
<td>1.618</td>
<td>0.922</td>
<td>2.540</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Pers</td>
<td>0.06</td>
<td>0.031</td>
<td>0.002</td>
<td>0.033</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Pers</td>
<td>0.12</td>
<td>0.078</td>
<td>0.004</td>
<td>0.082</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Pers</td>
<td>0.25</td>
<td>0.161</td>
<td>0.008</td>
<td>0.169</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Pers</td>
<td>0.56</td>
<td>0.403</td>
<td>0.019</td>
<td>0.422</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>1.12</td>
<td>0.813</td>
<td>0.038</td>
<td>0.852</td>
<td></td>
</tr>
<tr>
<td>100 small BP</td>
<td>Pers</td>
<td>12.61</td>
<td>1.303</td>
<td>0.102</td>
<td>1.405</td>
<td></td>
</tr>
</tbody>
</table>

### Two parallel applications putting only, 4-MB to 100-MB messages

**Table:** Very large message 9672-R56 CPU usage and elapsed time

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Elapsed time (s)</th>
<th>9672-R56 CPU (s/msg)</th>
<th>Application</th>
<th>Qmgr</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Nonpers</td>
<td>0.07</td>
<td>0.045</td>
<td>0.009</td>
<td>0.054</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Nonpers</td>
<td>0.58</td>
<td>0.250</td>
<td>0.113</td>
<td>0.363</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>5.90</td>
<td>1.611</td>
<td>2.319</td>
<td>3.930</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Pers</td>
<td>3.39</td>
<td>0.072</td>
<td>0.025</td>
<td>0.097</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Pers</td>
<td>16.14</td>
<td>0.442</td>
<td>0.117</td>
<td>0.559</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>85.06</td>
<td>2.117</td>
<td>0.620</td>
<td>2.737</td>
<td></td>
</tr>
</tbody>
</table>

### Two parallel applications getting only, 4-MB to 100-MB messages

**Table:** Very large message 9672-R56 CPU usage and elapsed time

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Elapsed time (s)</th>
<th>9672-R56 CPU (s/msg)</th>
<th>Application</th>
<th>Qmgr</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Nonpers</td>
<td>0.03</td>
<td>0.027</td>
<td>0.004</td>
<td>0.031</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Nonpers</td>
<td>0.28</td>
<td>0.155</td>
<td>0.049</td>
<td>0.204</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>2.38</td>
<td>0.940</td>
<td>0.716</td>
<td>1.656</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Pers</td>
<td>0.07</td>
<td>0.032</td>
<td>0.003</td>
<td>0.035</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Pers</td>
<td>0.25</td>
<td>0.159</td>
<td>0.017</td>
<td>0.176</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>1.80</td>
<td>1.040</td>
<td>0.062</td>
<td>1.102</td>
<td></td>
</tr>
</tbody>
</table>
Put/get of large messages, with parallel application putting/getting 1000-byte messages

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Large messages</th>
<th>Parallel 1000-byte msgs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Elapsed time (s)</td>
<td>Appl CPU (s/msg)</td>
</tr>
<tr>
<td>4</td>
<td>Nonpers</td>
<td>0.09</td>
<td>0.067</td>
</tr>
<tr>
<td>10</td>
<td>Nonpers</td>
<td>0.24</td>
<td>0.171</td>
</tr>
<tr>
<td>20</td>
<td>Nonpers</td>
<td>0.59</td>
<td>0.364</td>
</tr>
<tr>
<td>50</td>
<td>Nonpers</td>
<td>1.82</td>
<td>0.933</td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>4.66</td>
<td>2.040</td>
</tr>
<tr>
<td>4</td>
<td>Pers</td>
<td>1.44</td>
<td>0.102</td>
</tr>
<tr>
<td>10</td>
<td>Pers</td>
<td>3.58</td>
<td>0.252</td>
</tr>
<tr>
<td>20</td>
<td>Pers</td>
<td>6.89</td>
<td>0.538</td>
</tr>
<tr>
<td>50</td>
<td>Pers</td>
<td>17.31</td>
<td>1.351</td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>35.56</td>
<td>2.720</td>
</tr>
</tbody>
</table>

Parallel applications, one putting, one getting large messages

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>MQPUT</th>
<th>MQGET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Elapsed time (s)</td>
<td>Appl CPU (s/msg)</td>
</tr>
<tr>
<td>4</td>
<td>Nonpers</td>
<td>0.06</td>
<td>0.040</td>
</tr>
<tr>
<td>20</td>
<td>Nonpers</td>
<td>0.45</td>
<td>0.229</td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>3.88</td>
<td>1.331</td>
</tr>
<tr>
<td>4</td>
<td>Pers</td>
<td>1.71</td>
<td>0.074</td>
</tr>
<tr>
<td>20</td>
<td>Pers</td>
<td>7.36</td>
<td>0.367</td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>40.73</td>
<td>1.951</td>
</tr>
</tbody>
</table>
Single application putting with buffer pool too small, 100-MB messages

Table: Very large message 9672-R56 CPU usage and elapsed time

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Elapsed time (s)</th>
<th>9672-R56 CPU (s/msg)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Application</td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>53.83</td>
<td>8.954</td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>45.07</td>
<td>1.846</td>
</tr>
</tbody>
</table>

Single application getting with buffer pool too small, 100-MB messages

Table: Very large message 9672-R56 CPU usage and elapsed time

<table>
<thead>
<tr>
<th>Message size (MB)</th>
<th>Message persistence</th>
<th>Elapsed time (s)</th>
<th>9672-R56 CPU (s/msg)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Application</td>
</tr>
<tr>
<td>100</td>
<td>Nonpers</td>
<td>13.53</td>
<td>1.618</td>
</tr>
<tr>
<td>100</td>
<td>Pers</td>
<td>12.61</td>
<td>1.303</td>
</tr>
</tbody>
</table>

Distributed queuing

Single channel, persistent messages, with application removing messages at receiving end

Table: Very large message OS/390 9672-R56 CPU usage and elapsed time

<table>
<thead>
<tr>
<th>Msg size (MB)</th>
<th>Elapsed time (s/msg)</th>
<th>Sending end</th>
<th>Receiving end</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Qmgr CPU (s/msg)</td>
<td>Chin CPU (s/msg)</td>
</tr>
<tr>
<td>4</td>
<td>2.08</td>
<td>0.018</td>
<td>0.107</td>
</tr>
<tr>
<td>10</td>
<td>6.69</td>
<td>0.020</td>
<td>0.229</td>
</tr>
<tr>
<td>20</td>
<td>10.87</td>
<td>0.025</td>
<td>0.439</td>
</tr>
<tr>
<td>50</td>
<td>27.59</td>
<td>0.040</td>
<td>1.058</td>
</tr>
<tr>
<td>100</td>
<td>59.08</td>
<td>0.054</td>
<td>1.591</td>
</tr>
</tbody>
</table>
Single channel, nonpersistent messages, with application removing messages at receiving end

<table>
<thead>
<tr>
<th>Msg size (MB)</th>
<th>Elapsed time (s/msg)</th>
<th>Sending end</th>
<th>Receiving end</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Qmgr CPU (s/msg)</td>
<td>Chin CPU (s/msg)</td>
</tr>
<tr>
<td>4</td>
<td>1.30</td>
<td>0.015</td>
<td>0.096</td>
</tr>
<tr>
<td>10</td>
<td>2.78</td>
<td>0.014</td>
<td>0.232</td>
</tr>
<tr>
<td>20</td>
<td>3.53</td>
<td>0.015</td>
<td>0.454</td>
</tr>
<tr>
<td>50</td>
<td>10.38</td>
<td>0.017</td>
<td>1.227</td>
</tr>
<tr>
<td>100</td>
<td>21.87</td>
<td>0.014</td>
<td>2.802</td>
</tr>
</tbody>
</table>

Two parallel channels, persistent messages, with application removing messages at receiving end

<table>
<thead>
<tr>
<th>Msg size (MB)</th>
<th>Elapsed time (s/msg)</th>
<th>Sending end</th>
<th>Receiving end</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Qmgr CPU (s/msg)</td>
<td>Chin CPU (s/msg)</td>
</tr>
<tr>
<td>4</td>
<td>2.72</td>
<td>0.025</td>
<td>0.103</td>
</tr>
<tr>
<td>10</td>
<td>8.97</td>
<td>0.044</td>
<td>0.226</td>
</tr>
<tr>
<td>20</td>
<td>19.32</td>
<td>0.075</td>
<td>0.455</td>
</tr>
<tr>
<td>50</td>
<td>51.98</td>
<td>0.168</td>
<td>1.106</td>
</tr>
<tr>
<td>100</td>
<td>99.81</td>
<td>0.063</td>
<td>2.278</td>
</tr>
</tbody>
</table>

Two parallel channels, nonpersistent messages, with application removing messages at receiving end

<table>
<thead>
<tr>
<th>Msg size (MB)</th>
<th>Elapsed time (s/msg)</th>
<th>Sending end</th>
<th>Receiving end</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Qmgr CPU (s/msg)</td>
<td>Chin CPU (s/msg)</td>
</tr>
<tr>
<td>4</td>
<td>1.01</td>
<td>0.014</td>
<td>0.097</td>
</tr>
<tr>
<td>10</td>
<td>3.61</td>
<td>0.022</td>
<td>0.218</td>
</tr>
<tr>
<td>20</td>
<td>6.38</td>
<td>0.043</td>
<td>0.453</td>
</tr>
<tr>
<td>50</td>
<td>15.96</td>
<td>0.169</td>
<td>1.235</td>
</tr>
<tr>
<td>100</td>
<td>32.75</td>
<td>0.590</td>
<td>2.920</td>
</tr>
</tbody>
</table>
Test configuration

Hardware
• LPAR(s) of a 9672-RX6 processor, each with 5 dedicated CPUs (near equivalent to 9672-R56s). Each LPAR provided 5157.9 service units per second.
• Each LPAR had 2048 MB of central storage.
• LPARs shared a single IBM RAMAC Virtual Array 2 model T82 (RVA2-T82) DASD subsystem.
• Inter-LPAR TCP/IP connectivity via CTC connections.

Software
• OS/390 V2.7, IBM TCP/IP V2.7.
• MQSeries for OS/390 V2.1 with Very Large Message Support APAR PQ330000.
• Queue manager statistics trace enabled, 5 minute interval.
• Buffer pool 1 - 275000 pages (enough to hold 10 x 100-MB messages); buffer pool 2 - 10000 pages (too small to hold a single 100-MB message).
PART 4. PERFORMANCE

Two-tier - NT thin clients to OS/390 queue manager

Summary
For capacity planning purposes the total CPU costs for the TCPIP, CHINIT, queue manager and reply program address spaces are:

<table>
<thead>
<tr>
<th>Message length</th>
<th>Message persistence</th>
<th>9672-R56 CPUmillisecs / trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>Persistent</td>
<td>4.8</td>
</tr>
<tr>
<td>1KB</td>
<td>Non persistent</td>
<td>3.2</td>
</tr>
<tr>
<td>10KB</td>
<td>Persistent</td>
<td>6</td>
</tr>
<tr>
<td>10KB</td>
<td>Non persistent</td>
<td>4</td>
</tr>
<tr>
<td>25KB</td>
<td>Persistent</td>
<td>8.5</td>
</tr>
<tr>
<td>25KB</td>
<td>Non persistent</td>
<td>6</td>
</tr>
</tbody>
</table>

Introduction
This chapter contains the results of Windows NT thin client driven request/reply measurements for MQSeries for OS/390 V2.1 with APAR PQ30967. Results using V5.2 in non queue sharing group mode are not significantly different, for V5.2 shared queue usage see SupportPac MP1C.

All measurements employed the same simple request/reply model:

- The requester was represented by a number of Windows NT MQI thin client applications. Each client connected once then repeatedly MQPUT a single request message to a single request queue and issued MQGET (with wait) on a reply queue; both calls outside of syncpoint. The reply queue was either a single queue common to all clients (in which case the MQGET was by CorrelId) or a queue unique to each client application (using an MQGET without any MsgId/CorrelId). In order to maintain a desired transaction rate, an appropriate wait was inserted before putting the next request message.

- The clients were all connected to a single MQSeries for OS/390 V2.1 queue manager via TCP/IP server connection channels.

- Eight batch server applications were run swappable on the OS/390 host. All servers shared the same input request queue and retrieved (MQGET with wait) the next request message available. MQPUT1 was used to put a reply message of the same size and persistence as the request to the reply queue identified in the MQMD header, with a MsgId equal to the CorrelId of the request message. There was no business application logic. Again, both calls were outside of syncpoint. In terms of the number of forced log I/Os, this is an approximation to the case where, for example, a server participates in a two-phase commit performing an MQGET, a database update, then an MQPUT all within the same syncpoint, followed by a commit.

Throughout this chapter the term transaction is used to describe a request/reply message-pair. A client transaction begins with the MQPUT of the request message and ends when the MQGET-wait of the reply completes.

OS/390 CPU usage (shown as CPU per client transaction in the results below) was measured using RMF monitor I. Total CPU usage includes the CPU usage of the queue manager, channel initiator, server applications and TCP/IP. CPU usage for the swappable server applications excludes any Region Control Task (RCT) CPU time. RCT time is the overhead of swapping an address space and can be a significant cost. Elapsed transaction time and transaction rates were measured internally by the client/server applications.
Results summary

How do CPU costs and response times scale with increasing numbers of clients?

- Up to 1000 clients
  - Persistent messages: Initially, with between 100 and 300 clients, total CPU costs per transaction decreased with increasing clients. The decrease was more noticeable at lower transaction rates. As the number of clients increased above 300, the decrease in CPU costs became less apparent. The decrease was seen in the queue manager (due to the effect of increased log-write interleaving as the number of parallel I/Os increased), and also in the TCP/IP address space.
  - Nonpersistent messages: As with persistent messages, there was an initial decrease in total CPU costs per transaction as clients increased from 100 to 300, and again this was more noticeable at lower transaction rates. As the number of clients increased above 300, the CPU costs per transaction leveled out.

- 1000 to 9000 clients
  - Persistent messages: Total CPU costs per transaction remained approximately level as the number of clients increased up to 9000. The channel initiator costs increased slightly, but this was offset by the decreasing cost of the queue manager as log-write interleaving increased.
  - Nonpersistent messages: Total CPU costs per transaction showed a small, linear increase (1% to 2% per 1000 clients) as the number of clients increased up to 9000. This increase came entirely from the channel initiator.

- Response times generally showed a small increase with increasing numbers of clients as a result of a busier network and busier server applications.

How do CPU costs and response times scale with increasing transaction rate?

- Up to 1000 clients
  - Persistent messages: Total CPU costs per transaction generally decreased with increasing transaction rate up to 240 transactions/hour, then tended to level out. At the highest transaction rate used (900 transactions/hour per client), a small increase in total cost was observed with above 500 clients. The increase was seen mainly in the server applications and corresponded with a marked increase in elapsed transaction times and a high log DASD I/O rate. We conclude that our workload was now in log I/O constraint; the workload on your system will have its own point where log I/O becomes the constraint.
  - Nonpersistent messages: As with persistent messages, total CPU costs decreased with increasing transaction rate, and again this was more noticeable at the lower transaction rates and with smaller numbers of clients. Above a rate of around 480 transactions/hour per client, CPU costs leveled off.

- Response times again showed an increase with increasing transaction rate. The increase was more significant with persistent messages when the aggregate transaction rate exceeded around 100 transactions per second.

How do CPU costs and response times scale with increasing message size?

- Up to 1000 clients
  - With both persistent and nonpersistent messages, total CPU per transaction increased with increasing message size from 1 KB to 25 KB. The increase was generally linear in nature, with a similar rate of increase for persistent and nonpersistent messages. From 1 KB to 2 KB the increase in CPU per KB tended to be a little greater than from 2 KB upwards.
As expected, response times increased with increasing message size.

**How do multiple, unique reply queues compare to a single, common reply queue?**

In all cases, throughout the range of client numbers, a single common reply queue was cheaper than multiple (one per client) unique reply queues.

Prior to APAR PQ30967, which introduced enhancements to the internal MQ locking routines, a common reply queue was cheaper with small numbers of clients, but became more expensive than unique reply queues when the number of clients exceeded around 2000.

**How much CPU does any additional thin client MQI call use?**

The cost per TCP/IP thin client MQI call is of order 1.1 CPU millisecs (9672-R56) more than the basic cost of the MQI call.

Clearly it is best for performance to avoid MQI calls where possible. For instance, try to keep a queue open or use MQPUT1 rather than MQOPEN, MQPUT, MQCLOSE.

MQCONN and MQDISC are particularly expensive as they both establish/disestablish the client channel as well as connect/disconnect to the queue manager.

**How much CPU does a thin client MQCONN/MQDISC use?**

The cost of a thin client MQCONN/MQDISC pair is of order 14 CPU millisecs (9672-R56).

Clearly it is best for performance for a thin client to remain connected for as many business interactions, and thus for as many MQI calls, as possible.
CPU usage versus number of clients

Figure: Total OS/390 CPU usage v. number of clients. Persistent messages, common reply queue

Figure: Total OS/390 CPU usage v. number of clients. Nonpersistent messages, common reply queue
CPU usage versus transaction rate

**Figure:** Total OS/390 CPU usage v. transaction rate. Persistent messages, common reply queue

**Figure:** Total OS/390 CPU usage v. transaction rate. Nonpersistent messages, common reply queue
CPU usage versus message size

Figure: Total OS/390 CPU usage v. message size. Persistent messages, common reply queue

Figure: Total OS/390 CPU usage v. message size. Nonpersistent messages, common reply queue
Common compared to unique reply queues

Figure: Total OS/390 CPU. Common and Unique reply queues. Persistent messages. Up to 1000 clients

Figure: Total OS/390 CPU. Common and Unique reply queues. Nonpersistent messages. Up to 1000 clients
Figure: Total OS/390 CPU. Common and Unique reply queues. Persistent messages. Up to 9000 clients

Figure: Total OS/390 CPU. Common and Unique reply queues. Nonpersistent messages. Up to 9000 clients
Transaction round-trip times

Figure: Transaction response times v. number of clients. Common reply queue. 1KB messages. Up to 1000 clients

Figure: Transaction response times v. number of clients. Common reply queue. 1KB messages. Up to 9000 clients
Test configuration

Host
- LPAR of a 9672-RX6 processor with 5 dedicated CPUs (near equivalent to a 9672-R56)
- RVA2-T82 DASD subsystem
- OS/390 V2.7, IBM TCP/IP V2.7
- MQSeries for OS/390 V2.1 with APAR PQ30967
- TCP/IP connectivity via an OSA adapter connected to a 155Mbit ATM hub

Client PC
- IBM PC300GL with 400MHz Pentium II processor, 384MB memory
- Windows NT V4.0
- MQSeries for Windows NT V5.1 (client only)
- TCP/IP connectivity via 16Mbit token-ring (indirectly connected to host ATM hub)

Queue Manager
- Request queue with INDXTYPE(NONE), unique reply queues with INDXTYPE(NONE), common reply queue with INDXTYPE(CORRELID). We would not have expected significantly different results had the unique reply queues also used INDXTYPE(CORRELID).
- Buffer pools large enough to avoid any page set I/O
- Three dual logs (1667 cylinders of 3390), dual BSDSs and archiving to DASD active. Logs were large enough to avoid log switches during test runs.
- CSQ6LOGP: OUTBUFF=4000, WRTHRSH=15
- CSQ6SYSP: LOGLOAD=500000

Channel Initiator
- Single Server-connection TCP/IP channel definition
- CSQ6CHIP ADAPS=30, DISPS=20
Three-tier - NT thin clients to NT queue manager to OS/390 queue manager

Introduction

This chapter compares the results of Windows NT thin client measurements in a three-tier scenario (described below) with those of the two-tier scenario.

MQSeries for OS/390 V2.1 with APAR PQ30967 was used on the OS/390 host, no significant difference is expected when using V5.2 with non-shared queues.

The three-tier measurements employed the following simple request/reply model:

- The requester was represented by a number of Windows NT MQI thin client applications. Each client connected once then repeatedly MQPUT a single request message to a single request queue and issued MQGET (with wait) on a reply queue; both calls outside of syncpoint. The reply queue was either a single queue common to all clients (in which case the MQGET was by CorrelId) or a queue unique to each client application (with an MQGET of any MsgId/CorrelId). In order to maintain a desired transaction rate, an appropriate wait was inserted before putting the next request message.

- The clients connected to a single MQSeries for Windows NT V5.1 queue manager. This Windows NT queue manager was connected to a single MQSeries for OS/390 V2.1 queue manager via a single clustered TCP/IP channel pair, defined with NPMSPEED(FAST), BATCHSZ(50) and BATCHINT(100). A remote queue definition was used to identify the single, non-clustered request queue residing on the OS/390 queue manager. The reply queues were normal, non-clustered queues.

- Eight batch server applications were run swappable on the OS/390 host. All servers shared the same input request queue and retrieved (MQGET with wait) the next request message available. MQPUT1 was used to put a reply message of the same size and persistence as the request to the Windows NT reply queue identified in the MQMD header, with a MsgId equal to the CorrelId of the request message. There was no business application logic. Again, both calls were outside of syncpoint. In terms of the number of forced log I/Os, this is an approximation to the case where, for example, a server participates in a two-phase commit performing an MQGET, a database update, then an MQPUT all within the same syncpoint, followed by a commit.

Throughout this chapter the term transaction is used to describe a request/reply message-pair. A client transaction begins with the MQPUT of the request message and ends when the MQGET-wait of the reply completes.

OS/390 CPU usage (shown as CPU per client transaction in the results below) was measured using RMF monitor I. Total CPU usage includes the CPU usage of the queue manager, channel initiator, server applications and TCP/IP. CPU usage for the swappable server applications excludes any Region Control Task (RCT) CPU time. RCT time is the overhead of swapping an address space and can be a significant cost. Elapsed transaction time and transaction rates were measured internally by the client/server applications.

Results summary

How do three-tier OS/390 CPU costs compare with two-tier?

- Persistent messages: Three-tier (single channel pair) OS/390 CPU costs were always less than the equivalent two-tier case (multiple server connection channels), the difference becoming greater as the three-tier costs decreased with increasing numbers of clients. A significant factor here was the achieved batch size which varied from 3 to 4 with 200 clients up to 10 with 1000...
clients. The difference between the common and unique reply queue three-tier scenarios are also due to differences in achieved batch sizes in the 200 to 600 client range.

- Nonpersistent messages: Three-tier CPU costs were again always less than the two-tier case. The decrease in three-tier costs with increasing clients was less significant and corresponded with a smaller range of achieved batch sizes (3.5 to 6 with 200 clients up to 6.5 with 1000 clients).

Achieved batch sizes, the average number of messages in each channel batch, was determined from the MSGS and BATCHES values returned by the DISPLAY CHSTATUS command issued at the start and end of each measurement interval for both sender and receiver channels.

**How do three-tier response times compare with two-tier?**

- Persistent messages: Three-tier response times were approximately ten times greater than the equivalent two-tier case (for example 400 ms compared with 40 ms), and increased with the number of clients, whereas two-tier times remained more constant. There were two factors contributing to the higher three-tier persistent message response times:
  a. A channel batch interval of 100 milliseconds meant that, for example, the first message in an achieved batch of 10 would be delayed by up to 0.1 seconds. The average delay for 10 transactions (10 messages in on a receiver channel and 10 msgs out on a sender channel) would be of order 0.1 seconds. Response time could be reduced by decreasing the batch interval at the cost of an increase in CPU. The CPU cost is unlikely to be greater than that for two-tier even with a zero batch interval.
  b. Logging of every message occurred twice, once on the middle-tier Windows NT queue manager, where log I/Os were relatively slow, and once on the OS/390 queue manager.

- Nonpersistent messages: The three-tier scenario had
  - channels defined with NPMSPEED(FAST), so nonpersistent messages did not have to wait for the end of batch before they became available,
  - did not require any logging,
  - employed a faster TCP/IP network (100Mbit ethernet compared with 16Mbit token-ring in the two-tier case). Consequently, three-tier response times were lower than the equivalent two-tier case, although they did show a small increase with increasing number of clients.
OS/390 CPU usage

**OS/390 CPU usage - Two-tier vs. three-tier**
*Persistent 1KB msgs - 120 trans/hour/client*

![Graph showing CPU usage comparison between two-tier and three-tier systems for persistent 1KB messages.]

Figure: Total OS/390 CPU usage, 2-tier v. 3-tier. Persistent 1KB messages, 120 trans/hour/client.

**OS/390 CPU usage - Two-tier vs. three-tier**
*Nonpersistent 1KB msgs - 120 trans/hour/client*

![Graph showing CPU usage comparison between two-tier and three-tier systems for nonpersistent 1KB messages.]

Figure: Total OS/390 CPU usage, 2-tier v. 3-tier. Nonpersistent 1KB messages, 120 trans/hour/client.
Transaction response time

Figure: Transaction response time, 2-tier v. 3-tier. Persistent 1KB messages, 120 trans/hour/client.

Figure: Transaction response time, 2-tier v. 3-tier. Nonpersistent 1KB messages, 120 trans/hour/client.
Test configuration

OS/390 Host
- LPAR of a 9672-RX6 processor with 5 dedicated CPUs (near equivalent to a 9672-R56)
- RVA2-T82 DASD subsystem
- OS/390 V2.7, IBM TCP/IP V2.7
- MQSeries for OS/390 V2.1 with APAR PQ30967
- TCP/IP connectivity via an OSA adapter connected to a 155Mbit ATM hub

Server PC
- IBM Netfinity 5500-M20 with four 500MHz Pentium III Xeon processors, 2GB memory
- Windows NT Server V4.0
- MQSeries for Windows NT V5.1
- TCP/IP connectivity via 100Mbit ethernet (indirectly connected to host ATM hub)

Client PC
- IBM PC300GL with 400MHz Pentium II processor, 384MB memory
- Windows NT V4.0
- MQSeries for Windows NT V5.1 (client only)
- TCP/IP connectivity via 100Mbit ethernet (directly connected to server PC)

OS/390 Queue Manager
- Buffer pools large enough to avoid any page set I/O
- Three dual logs (1667 cylinders of 3390), dual BSDSs and archiving to DASD active. Logs were large enough to avoid log switches during test runs.
- CSQ6LOGP: OUTBUFF=4000, WRTHRSH=15
- CSQ6SYSP: LOGLOAD=500000

OS/390 Channel Initiator
- Clustered TCP/IP channel pair to the Windows NT queue manager, with NPMSPEED(FAST), BATCHSZ(50), BATCHINT(100)
- CSQ6CHIP ADAPS=30, DISPS=20

Windows NT Queue Manager
- Request queue with INDXTYPE(NONE), unique reply queues with INDXTYPE(NONE), common reply queue with INDXTYPE(CORRELID). We would not have expected significantly different results had the unique reply queues also used INDXTYPE(CORRELID).
MQSeries/ IMS bridge performance

There is no known significant change in the performance of the IMS bridge between MQSeries for OS/390 V5.2 and MQSeries for OS/390 V2.1. These measurements use MQSeries for OS/390 V5.2 unless otherwise stated.

Overview of the IMS bridge

The MQSeries-IMS bridge, known as the IMS bridge, provides the capability to schedule IMS transactions directly on IMS from an MQSeries message without the need to use an IMS trigger monitor. The IMS bridge, which is a component of the MQSeries for OS/390 queue manager, communicates with IMS using the IMS Open Transaction Manager Access (OTMA) service: The IMS bridge is an OTMA client.

When a legacy IMS transaction is driven from a 3270 type screen, any data entered on the screen is made available to the transaction by the IMS GU call. The transaction sends its response back to the terminal using the IMS ISRT call.

An MQSeries application can cause the same transaction to be scheduled by using the IMS bridge. An MQSeries 'request' message destined for IMS, typically with an MQIIH header, is put to an IMS bridge queue. The message is retrieved from the IMS bridge queue by the queue manager and sent to IMS over OTMA logical connections called transaction pipes or tpipes, where the MQSeries message data becomes input to the IMS GU call. The data returned by the ISRT call will be put into the reply-to queue, where the MQSeries application can retrieve it using a standard MQGET call.

This sequence of events is a typical use of the IMS bridge and forms the basis of the measurements presented in this chapter. The MQSeries request message and its associated reply are referred to as a message-pair in the rest of this chapter.

The following sections contain the results of IMS bridge measurements for MQSeries for OS/390 Version 5.2. Although CPU usage figures are given, the main focus is on the message-pair maximum throughput rates observed in several request/reply models.

The measurements are presented in three sections:

- **IMS bridge maximum throughput** which, as the title suggests, deals mainly with IMS Bridge maximum throughput in a variety of requester scenarios using both CICS and TSO/batch.

- **IMS Bridge thin client driven steady workload** reports IMS bridge CPU usage in a Windows NT thin client environment with a steady request-reply message-pair workload.

- **Historical IMS bridge measurements**

Summary of throughput results

Nonpersistent messages

Nonpersistent IMS bridge message throughput is typically constrained by CPU, especially if IMS logging is kept to a minimum by using nonrecoverable IMS transactions. If the system becomes CPU bound, increasing the number of bridge queues (and hence the number of OTMA tpipes) will not improve throughput. Total CPU usage of 99% was observed when using 2 tpipes or more during the nonpersistent message measurements.

The best results were seen in a TSO/batch driven scenario where the MQSeries queue manager and the IMS system were on separate OS/390 images, XCF linked via a coupling facility (ICS CF links slightly outperformed CBS CF links).

Running on two 3-engined 9672-X37 processors (see following chart) we achieved the order of:
• 1450 1000-byte nonpersistent message-pairs per second.

Running on a single 3-engined 9672-X37 processor (see preceding chart), we achieved the order of:
- **1070** 1000-byte nonpersistent message-pairs per second.
- **790** 5000-byte nonpersistent message-pairs per second.
- **700** 10,000-byte nonpersistent message-pairs per second.

![MQ IMS Bridge (CICS driven): Throughput](image)

Driving the IMS Bridge using CICS transactions which process a single request/reply message-pair, with queue manager, CICS and IMS on a single OS/390 image running on a 3-engined 9672-X37 processor, we achieved in the order of:

- **550** 1000-byte nonpersistent message-pairs per second with a single CICS region.
- **580** 1000-byte nonpersistent message-pairs per second with 4 CICS regions.

If each CICS transaction processes 100 message-pairs, throughput will increase to:

- **900** 1000-byte nonpersistent message-pairs per second with a single CICS region.

Network constraints prevented the measurement of maximum throughput in the Windows NT thin client scenario in section "IMS Bridge thin client driven steady workload". However, if it is assumed that once any network constraints are removed, CPU becomes the limiting factor as it was in the measurements in section "IMS bridge maximum throughput". Then a theoretical thin client maximum throughput may be derived. Dividing the maximum CPU usage seen in the throughput measurements (99%) by the CPU used per thin client message-pair gives a maximum throughput of approximately:

- **650** 1000-byte nonpersistent message-pairs per second
This theoretical maximum is also applicable to remote requester applications (connected to remote queue managers) sending messages over normal MQSeries channels where the OS/390 CPU costs are comparable.

**Persistent messages**

When persistent messages are used, throughput is constrained by DASD I/O, to both MQSeries and IMS log datasets. Message-pair throughput may be increased, up to a point, by adding more tpipes (8 tpipes appeared optimal in these measurements). More tpipes give increased parallelism within the IMS bridge and thus more efficient MQSeries log I/O writes.

With hardware similar to that used in these measurements, you can expect to drive the IMS bridge at around:

- **700** 1000-byte persistent message-pairs per second using IMS commit mode 1
- **390** 1000-byte persistent message-pairs per second using IMS commit mode 0

These results were obtained in a TSO/batch driven scenario on a single MVS image running on a 3-engined 9872-X37 processor.

Although the cost per message-pair of using a thin client or channel driven scenario is slightly greater, as the constraint is I/O and not CPU it is reasonable to expect that the best IMS bridge persistent message-pair throughput seen in these measurements could also be achieved by the MQSeries channel initiator with thin client requester applications or with remote requester applications sending messages over normal MQSeries channels.

![MQ IMS Bridge (Batch driven): Throughput](image)
**IMS bridge maximum throughput**

**Overview**

MQSeries messages destined for IMS, typically with an MQIIH header, are retrieved from an IMS bridge queue and sent to OTMA over logical connections called transaction pipes or tpipes. Each IMS bridge queue has up to two tpipes associated with it, one for commit mode 0 (commit_then_send) messages and another for commit mode 1 (send_then_commit) messages. Thus, when all messages use the same commit mode, the number of active tpipes is equal to the number of IMS bridge queues.

In the test measurements presented below, two IMS transaction types were used: Full-function, recoverable (i.e. with INQUIRY=(YES,RECOVER) specified) for all persistent MQSeries messages, and full-function, nonrecoverable (INQUIRY=(YES,NORECOVER)) for most nonpersistent messages. Fastpath (EMH) IMS transactions were not measured.

In recoverable transactions the entire message data is written to the IMS log at every GU and ISRT call. However, in a nonrecoverable transaction message data is not included in the log write, thus a higher message throughput rate might be expected.

The amount of On the IMS system used for the measurements the IMS data logged when using commit mode 1 was observed to be:

Nonrecoverable transaction

- GU call: approximately 1000 bytes logged
- ISRT call: approximately 1000 bytes logged

Recoverable transaction

- GU call: approximately (1000 + message size) bytes logged
- ISRT call: approximately (1000 + message size) bytes logged

**Test configuration**

IMS bridge throughput measurements were taken with MQSeries for OS/390 Version 5.2 and IMS/ESA Version 7.1. The MQSeries queue manager was not in a Queue Sharing Group (QSG) and therefore did not use any shared queues. In all measurements, IMS OTMA security was disabled, i.e. command /SEC OTMA NONE was issued.

In most scenarios the queue manager, IMS and the requester applications (including CICS where applicable) were all run on a single OS/390 image running OS/390 Version 2.10. CICS transactions were started by TPNS which also ran on the same OS/390 image. Measurements were also taken with the queue manager and driving applications on one OS/390 image and IMS on another OS/390 image (both on the same hardware and both OS/390 Version 2.10).

The target IMS transactions, full-function and either recoverable or nonrecoverable, were run in four message processing regions (MPRs). Each transaction simply retrieved a request message with a DL/I GU call, returned a reply of the same size with an ISRT call and terminated, i.e. there was no business application logic. All MQSeries request messages, MQPUT to IMS Bridge queues, contained an MQIIH header.

Three requester/reply scenarios were used, driven by either CICS MQI transactions or TSO/batch MQI applications:
• CICS system with a single MQOPEN/MQPUT/MQGET/MQCLOSE per CICS transaction.
  ▪ 1 and 4 CICS regions
  ▪ 1000-byte nonpersistent messages and full-function nonrecoverable transactions only

• CICS system with a loop of 100 MQOPEN/MQPUT/MQGET/MQCLOSE calls per CICS transaction.
  ▪ 1 CICS region
  ▪ 1000-byte nonpersistent messages and full-function nonrecoverable transactions only

• Eight batch applications with a single MQOPEN, repeated MQPUT/MQGET calls, then a single MQCLOSE.
  ▪ 1 and 2 MVS images
  ▪ 1000, 5000 and 10,000-byte nonpersistent, 1000-byte persistent messages
  ▪ Full-function nonrecoverable and recoverable transactions (with nonpersistent messages)
  ▪ Commit mode 0 and 1 (with persistent messages)

In the CICS scenarios CICS transactions were started from either 8 or 50 TPNS simulated terminals with zero think time between transactions. The transaction logic was as follows,

• Initialize
• Loop 1 or 100 times:
  ▪ MQOPEN IMS bridge and reply queues
  ▪ MQPUT a request message, with an MQIILH header, to an IMS bridge queue.
  ▪ COMMIT
  ▪ MQGET-wait on a reply queue unique to the CICS terminal.
  ▪ COMMIT
  ▪ MQCLOSE
  ▪ End of loop
  ▪ Terminate

In the single MQOPEN/MQCLOSE batch scenario, eight batch applications were run, each behaving as follows,

• Initialize
• MQCONN
• MQOPEN IMS bridge and reply queues
• Loop forever:
  ▪ MQPUT a request message, with an MQIILH header, to an IMS bridge queue.
  ▪ COMMIT
  ▪ MQGET-wait on a reply queue unique to the batch application.
  ▪ COMMIT
  ▪ End of loop
• MQCLOSE
• MQDISC
• Terminate

In the measurements that follow, an MQPUT of a request message followed by a completed MQGET-wait of the reply is referred to as a message-pair.

In all scenarios, messages were either persistent (IMS commit mode 0) or nonpersistent (commit mode 1) and of size 1000, 5000 or 10,000 bytes. When more than one IMS bridge queue was employed, their use was evenly distributed amongst the CICS terminals or batch applications. As all messages in a given measurement specified the same commit mode, each IMS bridge queue corresponded to a single tpipe between the IMS bridge and OTMA.
Hardware configuration

All measurements were taken on all or some part of a parallel sysplex. This consisted of three OS/390 Release 2.10 systems each running on a 3 dedicated engine LPAR of a 9672-XZ7 with 2GB of real and 1GB of extended storage. A 3-engined LPAR of a single 9672-XZ7 processor is near equivalent to a 9672-X37 processors, and is referred to as a 9672-X37 throughout the rest of this chapter. The coupling facility (CF) ran in a 2 dedicated engine LPAR of the same XZ7. (So one engine of the XZ7 was not used in this plex).

Each OS/390 system was connected to the CF by 4 CFS links, 1 CBS link and 3 ICS links. These were configured such that only one type was online at any given time. The CF was only employed (by the IMS Bridge and OTMA, via XCF) in the measurements with two OS/390 images.

Queue manager and IMS log datasets were located on separate logical 3390 volumes, all with DASD fastwrite switched on and typically able to achieve I/O response times of 1 or 2 milliseconds. These 3390s were physically part of a ESS 2105-E20 DASD subsystem with feature 2121 (9.1GB disks, the fastest).

Measurement results

In general, nonpersistent message scenarios tend to be limited by CPU, particularly when nonrecoverable IMS transactions limit the amount of IMS logging required. When the system is CPU bound, the addition of more tpipes will not improve throughput.

On the other hand, persistent message scenarios will be constrained by log I/Os (to both queue manager and IMS logs), not by CPU. In this case increasing the number of tpipes increases parallelism within the IMS bridge, including that of log I/O writes, and so one can expect throughput to increase accordingly.

The tables in the following sections give the maximum achieved transaction throughput and the CPU utilization.

All message-pair CPU measurements are those reported by RMF and are subject to a typical capture ratio of 0.90, i.e.

\[ \frac{\text{RMF-reported total CPU}}{\text{Hardware-reported total CPU}} = 0.90 \]

Message-pair throughput rates were the aggregated rates reported by each TSO/batch application, or the number of CICS transactions executions per second reported by RMF depending on the scenario.

Note that typically four or more IMS log switches and several IMS checkpoints occurred during the measurements.

Nonpersistent messages

TSO/batch driven, with queue manager and IMS running on separate OS/390 images
The best nonpersistent message throughput results were seen when the queue manager and IMS were running on separate OS/390 images, thus alleviating the constraint on CPU. Using 3 ICS CF links and 8 tpipes, a 1000-byte message throughput of 1455 message-pairs per second was observed (1358 message-pairs per second with 1 CBS CF link), compared to 1074 message-pairs per second when run on a single OS/390 image. The constraint here was I/Os to the IMS logs. Throughput increased with the number of tpipes, tailing off at 4 to 8 tpipes, whilst total CPU usage per message-pair remained roughly constant, thus CPU utilization increased up to 92% on the queue manager processor and 74% on the IMS processor. Message-pair CPU usage included that of XCF from both images.

TSO/batch driven, with queue manager and IMS running on a single OS/390 image
In the TSO/batch driven scenarios which ran on a single OS/390 image, 1000-byte message throughput rates peaked at 1074 message-pairs per second with 8 tpipes. As the constraint here was clearly CPU, hitting over 99% with 2 or more tpipes, increasing the number of tpipes above 2 had little effect on throughput. It is also apparent that CPU constraints masked any IMS logging benefit in using nonrecoverable over recoverable IMS transactions, as the achieved 1000-byte message throughput rates were very similar for both. This may not have been the case if recoverable transactions had been used with the larger message sizes.

### 8 Batch Jobs - 1000-byte nonpersistent msgs - Full-function, nonrecoverable IMS transaction

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Tpipes</th>
<th>Msg-pair rate (pps)</th>
<th>9672-X37 CPU (ms/tran)</th>
<th>Processor Capture</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>954</td>
<td>1.13 0.40 0.81 0.17</td>
<td>2.51</td>
<td>89.4%</td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>1047</td>
<td>1.12 0.43 0.83 0.19</td>
<td>2.57</td>
<td>99.5%</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>1068</td>
<td>1.09 0.42 0.82 0.19</td>
<td>2.52</td>
<td>99.6%</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>1074</td>
<td>1.08 0.42 0.82 0.19</td>
<td>2.51</td>
<td>99.6%</td>
</tr>
</tbody>
</table>

### 8 Batch Jobs - 5000-byte nonpersistent msgs - Full-function, nonrecoverable IMS transaction

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Tpipes</th>
<th>Msg-pair rate (pps)</th>
<th>9672-X37 CPU (ms/tran)</th>
<th>Processor Capture</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>1</td>
<td>683</td>
<td>1.51 0.54 1.00 0.21</td>
<td>3.25</td>
<td>84.9%</td>
</tr>
<tr>
<td>5000</td>
<td>2</td>
<td>777</td>
<td>1.55 0.56 1.03 0.22</td>
<td>3.36</td>
<td>99.0%</td>
</tr>
<tr>
<td>5000</td>
<td>4</td>
<td>794</td>
<td>1.52 0.55 1.01 0.22</td>
<td>3.30</td>
<td>99.5%</td>
</tr>
<tr>
<td>5000</td>
<td>8</td>
<td>796</td>
<td>1.51 0.55 1.02 0.22</td>
<td>3.30</td>
<td>99.6%</td>
</tr>
</tbody>
</table>

### 8 Batch Jobs - 10,000-byte nonpersistent msgs - Full-function, nonrecoverable IMS transaction

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Tpipes</th>
<th>Msg-pair rate (pps)</th>
<th>9672-X37 CPU (ms/tran)</th>
<th>Processor Capture</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>1</td>
<td>630</td>
<td>1.67 0.59 1.20 0.24</td>
<td>3.71</td>
<td>90.3%</td>
</tr>
<tr>
<td>10000</td>
<td>2</td>
<td>700</td>
<td>1.66 0.60 1.19 0.26</td>
<td>3.71</td>
<td>99.1%</td>
</tr>
<tr>
<td>10000</td>
<td>4</td>
<td>707</td>
<td>1.65 0.60 1.19 0.25</td>
<td>3.70</td>
<td>99.5%</td>
</tr>
<tr>
<td>10000</td>
<td>8</td>
<td>703</td>
<td>1.64 0.61 1.20 0.26</td>
<td>3.71</td>
<td>99.5%</td>
</tr>
</tbody>
</table>

### 8 Batch Jobs - 1000-byte nonpersistent msgs - Full-function, recoverable IMS transaction

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Tpipes</th>
<th>Msg-pair rate (pps)</th>
<th>9672-X37 CPU (ms/tran)</th>
<th>Processor Capture</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>961</td>
<td>1.14 0.41 0.84 0.16</td>
<td>2.54</td>
<td>91.5%</td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>1039</td>
<td>1.12 0.43 0.85 0.18</td>
<td>2.57</td>
<td>99.3%</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>1064</td>
<td>1.09 0.42 0.84 0.17</td>
<td>2.52</td>
<td>99.4%</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>1070</td>
<td>1.09 0.42 0.85 0.17</td>
<td>2.53</td>
<td>99.5%</td>
</tr>
</tbody>
</table>

### CICS driven, with queue manager, IMS and CICS running on a single OS/390 image

In the CICS driven scenario with a single CICS region and 50 simulated terminals, each transaction processing only a single message, maximum throughput of 556 1000-byte message-pairs per second was achieved with a single tpipe. With additional tpipes the throughput gradually tailed off to 536 message-pairs per second with 8 tpipes. This maximum figure was somewhat lower than the scenario in which 100 message-pairs were processed per transaction for several reasons: Firstly, the overhead of starting/stopping a CICS transaction was incurred for every message-pair. Secondly, as well as the CPU overhead of creating new queue manager control blocks, there was some degree of serialization in the queue manager process that acquired the necessary storage, thus adding some lock-wait time to the overhead. In addition, all 50 terminals were vying for the 8 CICS MQSeries adapters. The additional CPU costs of this scenario can be seen in the CPU per message-pair values for the CICS region and the queue manager.

A small improvement in throughput was seen with 4 CICS regions, each with 8 terminals: A peak of 587 1000-byte message-pairs per second was achieved, this time with 8 tpipes. With only 8 terminals per CICS region, contention for the 8 CICS MQSeries adapters was removed hence the small throughput improvements.
In the scenario where 100 message-pairs were processed in each transaction, a much improved peak throughput of 905 1000-byte message-pairs per second was observed with 8 tpipes. In this case transaction startup costs were incurred less frequently. Although not shown in the table below, 16 tpipes were also used in this scenario, but with little improvement on throughput (917 message-pairs per second.)

It should be noted that in the CICS driven scenarios TPNS was running on the same OS/390 image. TPNS CPU usage is not included in the tables below, hence the calculated "CPU used by test" figures where each CICS transaction processes only 1 message-pair (and so where TPNS is busiest) fall some way short of the observed "Processor CPU busy" values.

### Single CICS, 50 terminals - 1 msg-pair per CICS tran - 1000-byte non-persistent msgs

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Types</th>
<th>Msg-pair rate (l/s)</th>
<th>9672-X37 CPU (ms/tran)</th>
<th>Processor CPU busy (%)</th>
<th>Capture CPU used (%)</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>556</td>
<td>1.45</td>
<td>1.57</td>
<td>0.84</td>
<td>0.18</td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>545</td>
<td>1.39</td>
<td>1.54</td>
<td>0.83</td>
<td>0.19</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>545</td>
<td>1.38</td>
<td>1.51</td>
<td>0.82</td>
<td>0.19</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>536</td>
<td>1.37</td>
<td>1.46</td>
<td>0.82</td>
<td>0.19</td>
</tr>
</tbody>
</table>

### Single CICS, 50 terminals - 100 msg-pairs per CICS tran - 1000-byte non-persistent msgs

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Types</th>
<th>Msg-pair rate (l/s)</th>
<th>9672-X37 CPU (ms/tran)</th>
<th>Processor CPU busy (%)</th>
<th>Capture CPU used (%)</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>829</td>
<td>1.26</td>
<td>0.98</td>
<td>0.83</td>
<td>0.18</td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>859</td>
<td>1.24</td>
<td>0.91</td>
<td>0.84</td>
<td>0.19</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>898</td>
<td>1.21</td>
<td>0.81</td>
<td>0.82</td>
<td>0.19</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>905</td>
<td>1.19</td>
<td>0.81</td>
<td>0.82</td>
<td>0.19</td>
</tr>
</tbody>
</table>

### 4 CICS, 32 terminals (8 per CICS) - 1 msg-pair per CICS tran - 1000-byte non-persistent msgs

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Types</th>
<th>Msg-pair rate (l/s)</th>
<th>9672-X37 CPU (ms/tran)</th>
<th>Processor CPU busy (%)</th>
<th>Capture CPU used (%)</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>569</td>
<td>1.47</td>
<td>1.76</td>
<td>0.85</td>
<td>0.19</td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>580</td>
<td>1.43</td>
<td>1.72</td>
<td>0.84</td>
<td>0.19</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>581</td>
<td>1.42</td>
<td>1.72</td>
<td>0.84</td>
<td>0.19</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>587</td>
<td>1.40</td>
<td>1.70</td>
<td>0.84</td>
<td>0.19</td>
</tr>
</tbody>
</table>
Persistent messages

In the TSO/batch driven scenarios which ran on a single OS/390 image, 1000-byte persistent message throughput reached 391 message-pairs per second using commit mode 0, and 718 message-pairs per second using commit mode 1, both with 8 tpipes. Total CPU only reached 79%, whilst MQSeries log I/O rates in excess of 400 I/O per second were observed: As expected, the constraint on persistent message throughput was therefore log I/O. Throughput and response times improved rapidly up to 8 tpipes because with more tpipes, and hence more IMS bridge queues, parallel MQPUTs and MQGETs allow for more efficient MQSeries log I/O.

The throughput of persistent MQSeries messages invoking recoverable IMS transactions in commit mode 0 (commit_then_send) messages is lower than that of persistent MQSeries messages invoking recoverable IMS transactions in commit mode 1 (send_then_commit) because of the serialization and additional logging required to support once and once only delivery of the message.

### 8 Batch Jobs - 1000-byte persistent msgs - Full-function, recoverable IMS transaction - Commit mode 0

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Tpipes</th>
<th>Msg-pair rate (%)</th>
<th>Omgr</th>
<th>Batch</th>
<th>IMS Ctrl</th>
<th>IMS Mprs</th>
<th>Total</th>
<th>Processor CPU busy (%)</th>
<th>Capture rate (%)</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1.97</td>
<td>0.48</td>
<td>1.00</td>
<td>0.14</td>
<td>3.59</td>
<td>13.8%</td>
<td>0.85</td>
<td>11.4%</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>1.69</td>
<td>0.45</td>
<td>0.96</td>
<td>0.14</td>
<td>3.23</td>
<td>19.4%</td>
<td>0.89</td>
<td>17.4%</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>1.62</td>
<td>0.46</td>
<td>0.95</td>
<td>0.15</td>
<td>3.18</td>
<td>31.1%</td>
<td>0.91</td>
<td>29.1%</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>1.54</td>
<td>0.48</td>
<td>0.94</td>
<td>0.16</td>
<td>3.12</td>
<td>47.7%</td>
<td>0.91</td>
<td>44.6%</td>
<td></td>
</tr>
</tbody>
</table>

### 8 Batch Jobs - 1000-byte persistent msgs - Full-function, recoverable IMS transaction - Commit mode 1

<table>
<thead>
<tr>
<th>Msg size</th>
<th>No. of Tpipes</th>
<th>Msg-pair rate (%)</th>
<th>Omgr</th>
<th>Batch</th>
<th>IMS Ctrl</th>
<th>IMS Mprs</th>
<th>Total</th>
<th>Processor CPU busy (%)</th>
<th>Capture rate (%)</th>
<th>CPU used by test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1.66</td>
<td>0.45</td>
<td>0.83</td>
<td>0.14</td>
<td>3.07</td>
<td>18.7%</td>
<td>0.88</td>
<td>18.3%</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>2</td>
<td>1.61</td>
<td>0.46</td>
<td>0.84</td>
<td>0.15</td>
<td>3.07</td>
<td>34.0%</td>
<td>0.91</td>
<td>31.8%</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>1.51</td>
<td>0.50</td>
<td>0.85</td>
<td>0.17</td>
<td>3.03</td>
<td>56.3%</td>
<td>0.91</td>
<td>53.4%</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>1.48</td>
<td>0.40</td>
<td>0.84</td>
<td>0.17</td>
<td>2.88</td>
<td>78.6%</td>
<td>0.92</td>
<td>75.1%</td>
<td></td>
</tr>
</tbody>
</table>
**IMS Bridge thin client driven steady workload**

The measurements presented in this section are of a simple Windows NT thin client driven MQSeries-IMS Bridge request/reply scenario using just 1000-byte nonpersistent messages. Steady workloads were measured rather than maximum throughputs in order to avoid any effects caused by TCP/IP network bandwidth constraints or by other network traffic.

**Summary**

OS/390 CPU measurements were taken using Windows NT thin clients sending requests at a fixed rate per client (aggregate rates for all clients are shown in the table below). In range measured, CPU usage per message-pair was seen to decrease with an increasing number of clients, leveling off at around 300 clients.

If the total CPU used at the fixed workload rates is compared with the total CPU in the TSO/Batch maximum throughput tests in section "IMS bridge maximum throughput" above, a theoretical maximum thin client throughput may be estimated for nonpersistent messages as follows:

At 33.3 1000-byte nonpersistent message-pairs per second, the 500 thin clients required approximately 4.4 milliseconds of 9672-X37 CPU per message-pair (allowing for a capture ratio of 0.9). Total CPU was thus 33.3 x 4.4ms = 0.15 seconds of CPU, or 5% of 3 CPUs.

In the TSO/Batch driven scenario measurements of section "IMS bridge maximum throughput", up to 98% of 3 CPUs were used for 1000-byte nonpersistent messages. Thus, assuming no TCP/IP network constraints, you could expect a maximum thin client throughput of around (98 / 5) * 33.3, i.e. 650 1000-byte nonpersistent message-pairs per second.

**Test configuration**

IMS bridge CPU measurements were taken using a thin client driven requester/reply model, connecting via TCP/IP server-connection channels.

Between 100 and 500 thin client applications were run under Windows NT on a single IBM PC 300GL. Each client put nonpersistent MQIIH-prefixed messages outside of syncpoint at a fixed rate to one of 20 IMS bridge queues (and therefore 20 tpipes). Full-function recoverable IMS transactions and commit mode 1 (send_then_commit) were used throughout. Reply messages were retrieved from a single common reply queue.

The same hardware and software as in section "IMS bridge maximum throughput" above were used, with the MQSeries for OS/390 Version 5.2 queue manager and the IMS Version 7.1 system run on a single OS/390 image (OS/390 Version 2.10) on the equivalent of a 3-engined 9672-X37 processor. An OSA Express adapter connected OS/390 to a 155-Mbit ATM hub which in turn connected to a 100-Mbit Ethernet card on the PC via several intermediate hops.

The target IMS full-function recoverable transaction retrieved the message with a DL/I GU call and returned a reply of the same size with an ISRT call; there was no business application logic. As the transactions used were defined as recoverable, i.e. INQUIRY=(YES,RECOVER), each GU/ISRT call resulted in the entire message data being written to the IMS log.
Results

The following tables give client transaction response times in milliseconds and CPU usage in milliseconds per transaction for the queue manager (Qmgr), channel initiator (Chinit), IMS message processing regions (IMS Mprs), IMS control region (IMS Ctl) and TCPIP.

Windows NT thin client driven IMS Bridge workload

<table>
<thead>
<tr>
<th>Msg size</th>
<th>Number of thin clients</th>
<th>Msg-pair rate (/sec)</th>
<th>Round-trip time (sec)</th>
<th>9672-X37 CPU (ms/tran)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>100</td>
<td>6.7</td>
<td>0.017</td>
<td>0.99</td>
</tr>
<tr>
<td>1000</td>
<td>200</td>
<td>13.3</td>
<td>0.023</td>
<td>0.96</td>
</tr>
<tr>
<td>1000</td>
<td>300</td>
<td>20.0</td>
<td>0.027</td>
<td>0.96</td>
</tr>
<tr>
<td>1000</td>
<td>400</td>
<td>26.7</td>
<td>0.032</td>
<td>0.96</td>
</tr>
<tr>
<td>1000</td>
<td>500</td>
<td>33.3</td>
<td>0.033</td>
<td>0.95</td>
</tr>
</tbody>
</table>
**Historical IMS bridge measurements**

This contains MQSeries for MVS/ESA Version 1.1.4. CPU usage comparisons between applications using the IMS bridge and applications using the IMS trigger monitor, derived from material originally published in SupportPac MP12.

The same user application, running in batch, was used for both the IMS bridge and trigger monitoring processing.

Note: For these measurements, security checking was disabled for both MQSeries and OTMA.

**Overview of a transaction using the IMS bridge.**

An MQSeries application can cause a legacy IMS transaction to be scheduled by putting an appropriate MQSeries message to an IMS bridge queue. The MQSeries message data becomes input to the IMS GU call, and data returned by the ISRT call will be put into the reply-to queue, where the MQSeries application can retrieve it using a standard MQGET call.

**Flow of application using the IMS bridge:**

1. The user application puts a message to a queue specifying commit mode 0.
2. The application commits the message making it visible to other applications.
3. The user application issues an MQGET with wait option on reply-to queue.
4. The IMS bridge code, running in the queue manager address space, issues an MQGET of the message. The IMS bridge uses the OTMA interface to pass the information to IMS and waits for an acknowledgment.

The next two steps are independent and can occur in parallel:

5. The IMS bridge commits the get of the message.
6. IMS schedules the transaction in a message processing region (MPR).
7. The IMS transaction issues a GU call to obtain the information passed to it.
8. The application processes the message and issues an ISRT call to send the reply back to IMS.
9. IMS sends the reply to the IMS bridge (running in the queue manager address space) using the OTMA interface.
10. The IMS bridge uses the MQPUT1 call to put the message to the reply-to queue.
11. This message satisfies the outstanding MQGET request from the application program.
12. The application issues a commit on the message just got.

**Overview of a Trigger monitor application**

In releases of MQSeries prior to version 1.1.4, the only way to schedule an IMS transactions was to use the IMS trigger monitor. An MQSeries trigger monitor program running in a BMP waited for an MQSeries trigger message, it then scheduled an IMS transaction to run the MQSeries application.
Flow of an application using the trigger monitor:

1. The user application puts a message to a queue causing a trigger message to be produced.
2. The application commits the message making both the application message and the generated trigger messages available to other applications.
3. The application issues an MQGET call with the wait option on the reply-to queue.
4. The trigger message satisfies the MQGET call in the trigger monitor program.
5. The trigger monitor uses the IMS ISRT call to schedule the required transaction passing the queue manager name and the queue name.
6. The IMS transaction runs in a message processing region (MPR) and issues an IMS GU call to get the queue manager and queue names. The transaction issues the following calls:
   a. MQCONN to connect to the given queue manager. (1)
   b. MQOPEN of the passed queue name.
   c. MQGET of the user's message.
   d. MQPUT1 to put the reply message in the reply-to queue specified in the input message header.
   e. MQCLOSE of the input queue.
   f. MQDISC.
   g. The end of transaction causes IMS to perform commit processing which causes an MQSeries commit to be issued.
7. The reply message satisfies the outstanding MQGET with wait option in the user's application program.
8. The user's application commits the message just got.

Comparison of MQSeries calls between IMS bridge and trigger monitor applications

When using the IMS bridge application as described in "Flow of application using the IMS bridge" above, the following MQSeries calls are processed with the number of occurrences in brackets.

MQPUT(1) User application
MQPUT1(1) IMS bridge
MQGET(2) IMS bridge and user application
Commit(4) User application, IMS bridge (for MQGET), IMS bridge (for MQPUT1) and user application

When using the trigger monitor application described in "Flow of an application using the trigger monitor" above, the following MQSeries calls are processed with the number of occurrences in brackets:

MQCONN(1) IMS application
MQDISC(1) IMS application
MQPUT(2) User application and generated trigger messages
MQPUT1(1) IMS transaction
MQGET(3) Trigger monitor, IMS application and user application (3)

Commit(4) User application, trigger monitor, commit from IMS at end of transaction, and user application.

Note that if there are no messages on the queue for an MQGET with wait, then the task waits for a message. When the message arrives an additional MQGET request is reissued "under the covers" to get the message.

**CPU usage comparison between IMS bridge and IMS trigger monitor**

**Description of user application**

The IMS transactions were written in COBOL and used the IMS pseudo wait for input capability in IMS. This means that at the end of the transaction instead of terminating, it loops to the top and issues the IMS GU call again; if no IMS message is available, IMS suspends the transaction until there is more work for it to do. This eliminates the very expensive overhead of initialization and termination of the transaction.

The user application described in "Flow of application using the IMS bridge" and "Flow of an application using the trigger monitor" above was a requester/responder type application, running as a batch job and written in assembler language. The user application issues a request to a back-end transaction which processes the request and sends back a reply. The user application processing is described below.

- Determine the type of measurement from the supplied parameters. This included the use of persistent or non-persistent messages.
- Determine the output queue name to be used.
- Determine the input queue name to be used.
- Connect to the queue manager.
- Open the output queue.
- Open the input (reply-to) queue.
- Determine the before timer value.
- Do I = 1 to 1000
  - MQPUT message of length 1K
  - COMMIT
  - MQGET with wait option (reply message is 1K)
  - COMMIT
- End
- Determine after time value.
- Calculate after time minus before time.
- Display the average time per iteration of the loop.
- Close the input queue.
- Close the output queue.
- Disconnect from the queue manager.

The program was run 10 times in one job and at the end of each job step an RMF report was produced. Consistency checks were then made by checking values in the multiple reports.

**Application scenarios used**

The loop-back scenarios described below represent the cost when no IMS work is done, to show the relative costs of the IMS bridge.

**Scenario 1** IMS bridge using persistent messages

**Scenario 2** IMS bridge using non-persistent messages
Scenario 3 IMS trigger monitor using persistent messages

Scenario 4 IMS trigger monitor using non-persistent messages

Scenario 5 Loop back using persistent messages - the application puts a persistent message and immediately gets it.

Scenario 6 Loop back using non-persistent messages - the application puts a non-persistent message and immediately gets it.

Comparison of CPU time used

The table below shows the amount of ES/9000 820 CPU (in milliseconds) to process one complete request/response in the user application, or the cost of one IMS transaction when using TPNS.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>User application</th>
<th>Queue manager</th>
<th>Trigger monitor</th>
<th>IMS CNTL</th>
<th>IMS transaction</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1 - IMS bridge</td>
<td>2.0</td>
<td>6.3</td>
<td></td>
<td>3.7</td>
<td>0.4</td>
<td>12.4</td>
</tr>
<tr>
<td>persistent messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scenario 2 - IMS bridge</td>
<td>1.5</td>
<td>3.9</td>
<td></td>
<td>3.5</td>
<td>0.4</td>
<td>9.3</td>
</tr>
<tr>
<td>nonpersistent messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scenario 3 - IMS trigger</td>
<td>2.1</td>
<td>2.4</td>
<td>1.3</td>
<td>1.8</td>
<td>5.0</td>
<td>12.6</td>
</tr>
<tr>
<td>monitor persistent msgs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scenario 4 - IMS trigger</td>
<td>1.8</td>
<td>0.8</td>
<td>1.3</td>
<td>1.7</td>
<td>4.8</td>
<td>10.4</td>
</tr>
<tr>
<td>monitor nonpersistent msgs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scenario 5 - Loop back</td>
<td>1.5</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td>3.0</td>
</tr>
<tr>
<td>persistent messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scenario 6 - Loop back</td>
<td>1.1</td>
<td>0.4</td>
<td></td>
<td></td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>nonpersistent messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: Before each measurement was run an IMS SWITCH OLDS command and an MQSeries ARCHIVE LOG command were issued to prevent a log archive occurring during the measurements.

There is an increase in log prefix overhead from IMS V4 to V5 and also when using OTMA.
APPENDIX A.

Effect of LOGLOAD on restart time

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

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

Increase in startup time after abnormal shutdown

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

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

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

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

<table>
<thead>
<tr>
<th>Number of CICS transactions</th>
<th>Time between startup and CSQR001I</th>
<th>Time between CSQR001I and CSQR002I</th>
<th>Time between CSQR002I and CSQY022I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7 seconds</td>
<td>1 second</td>
<td>0.2 seconds</td>
</tr>
<tr>
<td>10000</td>
<td>7 seconds</td>
<td>59 seconds</td>
<td>0.2 seconds</td>
</tr>
</tbody>
</table>

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

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

A checkpoint is taken at the following times:

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

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

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

**Effect of the number of objects defined on startup time**

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

<table>
<thead>
<tr>
<th>Number of objects defined</th>
<th>Time between startup and CSQR001I</th>
<th>Time between CSQR001I and CSQR002I</th>
<th>Time between CSQR002I and CSQY022I</th>
</tr>
</thead>
<tbody>
<tr>
<td>140</td>
<td>7</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>4140</td>
<td>7</td>
<td>3.6</td>
<td>0.2</td>
</tr>
<tr>
<td>14484</td>
<td>7</td>
<td>7.6</td>
<td>0.2</td>
</tr>
</tbody>
</table>

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

The log manager is responsible for writing recovery information to the log data sets. This information is used to recover in the event of a failure or a request to roll back recoverable changes. Recoverable resources includes persistent messages and MQ objects. Nonpersistent messages are not recoverable and are not handled by the log manager; they are lost at system restart.

This appendix discusses only recoverable resources.

The log is conceptually a very long buffer. In practice the log is implemented using virtual storage and DASD. The position of information in this buffer is defined by the Relative Byte Address (RBA) from the start of the buffer.

Description of log manager concepts and terms

This section describes the concepts and terms used in this appendix. They are described more fully in the MQSeries for MVS/ESA System Management Guide.

- Each log buffer is 4096 bytes long and resides in virtual memory.
- The number of log buffers is determined from the OUTBUFF keyword of the CSQ6LOGP macro.
- When the log buffers fill, or an application issues a commit, the buffers are moved from virtual storage to log data sets, called the active log data sets. When the log records have been written, the log buffers can be reused.
- There are at least two active log data sets, which are used cyclically.
- Dual logging describes the situation where the log buffers are written to two log data sets. In the event of the loss of one data set, the other data set can be used. This facility is enabled with the TWOACTV keyword of the CSQ6LOGP macro.
- Single logging is when only one ring of active data sets are used.
- When an active log data set fills up, an archive log data set is allocated and the active log is copied to it. When the copying has completed, the active log data set can then be reused.
- A data set called the bootstrap data set (BSDS) records which RBA range is on which active or archive log. At system restart, the BSDS is used to identify which log data set to use first.
- You can have two copies of the BSDS data set, so in the event of the loss of one BSDS, the other can be used.
- When an active log is archived, the BSDS data sets are also archived.

Other terms used in this description

- The current log buffer is the log buffer that is being filled. When this buffer fills up, the next buffer is used and becomes the current log buffer.
- The logger is a task, running within the queue manager, that handles the I/O to log data sets.
- A log check request occurs while work is being committed. If the data up to the RBA has not been written out to disk, a request is made to the logger passing the RBA value, and the requester is suspended. The logger writes the data up to the RBA out to disk and resumes any tasks waiting for the logger. When the log check request completes, the data has been copied to disk and it can be used in the event of a failure. A log check is issued when:
  - A commit is issued.
  - A persistent message is put or got out of syncpoint.
  - An MQ object, such as a local queue, is defined, deleted or altered.
**Illustration of logging**

The following section gives a simplified view of the data that is logged when an application gets a persistent messages and issues a commit.

When a message is got, a flag is set to indicate that the message is no longer available. The change to the flag and information to identify the message within the page, along with information to identify the unit of work, are written to the log buffers. During the commit, "end of unit of work" information is written to disk and a log check request is issued with an RBA of the highest value used by the application.

**When does a write to the log data set occur?**

Log buffers are written to disk at the following times:

- When a log check request is issued. When the application is running under a syncpoint coordinator (for example, CICS/ESA) and has issued update requests to multiple resource managers (such as MQ requests) and recoverable CICS resources, the sync level 2 protocol is used at commit time. This causes two MQ log check requests, one for the PREPARE, and the other for the COMMIT verbs.

- If the number of filled log buffers is greater than or equal to the value of WRTHRSH specified in the CSQ6LOGP macro, a request is made to the logger to write out up to the RBA of the previous page.

- When all of the log buffers are in use and there are none free.

- When the system shuts down.

The logger writes up to 15 log buffers at a time to the log data sets, so 16 log buffers require at least two I/O requests, (but the buffers might be written out when other applications are issuing log check requests).

**How data is written to the active log data sets**

The current log buffer is the buffer that is currently being filled with data. When this buffer fills up, the next buffer is used and becomes the current log buffer.

**Single logging**

If the log check request specifies an RBA that is not in the current buffer, the logger writes up to and including the page containing the specified RBA.

If the log check request specifies an RBA that is in the current buffer:

- The logger writes any log buffers up to, but not including, the current buffer. Any I/O to the two data sets is performed in parallel.
- It writes the current log buffer to each log data set. The first time the current buffer is written to the log data sets, the I/O is performed in parallel. Any rewrite of the buffer writes to each log data set.
in series. This happens when a log check specifies an RBA in the same log buffer, or the log buffer is full and there is a new current log buffer. This is to maintain data integrity in case the update of the current log buffer on disk fails.

**Rule of thumb**

In effect, for a log check request with dual logging, the elapsed time for the write of the current page to the log data sets is the time required for two I/Os in series; all other log writes take the time for one log I/O.

**Interpretation of key log manager statistics**

Consider an application that gets two messages and issues a commit.

When a message is retrieved, a flag is set to indicate that the message is no longer available. The change to the flag and information to identify the message within the page, along with information to identify the unit of work, are written to the log buffers. The same happens for the second message. During the commit, "end of unit of work" information is written to disk and a log check request is issued with an RBA of the highest value used by the application.

If this is the only work running in the queue manager, the three log requests are likely to fit in one log buffer. The log manager statistics (described in the System Management Guide) would show the following:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>QJSTWRNW</td>
<td>Number of log writes with no wait</td>
<td>3</td>
</tr>
<tr>
<td>QJSTBFFL</td>
<td>Number of log pages used</td>
<td>1</td>
</tr>
<tr>
<td>QJSTBFWR</td>
<td>Number of calls to logging task</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Number of I/O to each log data set</td>
<td>1</td>
</tr>
</tbody>
</table>

In reality, more data than just one flag is logged and there might be more than one I/O involved. This is explained below.

**Detailed example of when data is written to log data sets**

Consider two applications, each putting a persistent message and issuing a commit. Assume that:

- Each message needs 16 log buffers to hold all of the data
- The WRTHRSH value in CSQ6LOGP is 20
- Dual logging is used
The following figure shows the log buffers used:

<table>
<thead>
<tr>
<th>Message 1</th>
<th>Message 2</th>
<th>Commit 1</th>
<th>Commit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1 B2 ... B15 B16</td>
<td>B17 B18 ... B31 B32</td>
<td>B33</td>
<td>B34</td>
</tr>
</tbody>
</table>

Where

- B1...B16 are the 16 log buffers for message 1
- B17...B32 are the 16 log buffers for message 2
- B33 is the log buffer for the commit of the first application
- B34 is the log buffer for the commit of the second application

In reality, each log buffer usually contains information from different applications, so an individual log buffer might contain information from message 1 and message 2.

If the interval between each MQPUT and the commit is relatively long compared to the time taken to do a disk I/O (for example, more than 20 milliseconds), the following happens:

1. The first message is put, buffers B1-B16 are filled.
2. When the second message is being put, and buffer B21 is about to be filled, because the number of full log buffers is greater than the value of WRTHRSH in CSQ6LOGP, this signals the logger to write out pages up to (but not including) the current buffer. This means that buffers B1-B20 are written out, buffers B1-15 in one I/O, and buffers B16-B20 in a second I/O.
3. When buffer B22 is being filled, the number of full log buffers is greater than WRTHRSH so a request is made to the logger, passing the RBA of page B21. Similarly, when writing B23 a request is made to the logger to write out buffer B22.
4. When the I/O to buffers B1-B15 has completed, these buffers are available for reuse, and so the number of full buffers falls below the value in WRTHRSH and no more requests are made to the logger.
5. When buffer B23 is being filled, the number of full log buffers is not greater than WRTHRSH, so a request is not made to the logger.
6. When the logger has finished processing the requests for buffers B1-15 and B16-20, it checks the work on its input queue. It takes the highest RBA found and writes up to that page to the data sets (so it would write out pages B21-B22). In practice, all of the buffers B23-B32 would be filled while the I/O of buffers B1-B15 is happening.
7. When commit 1 is issued, a log check is issued and buffers B23-B32 are written out in one I/O and buffer B33 (the current buffer) written out in a second I/O. The I/O for buffers B21-B32 is performed in parallel, and because this is the first time B33 has been written, the I/O is performed in parallel. The time taken for the commit is at least the time to perform two I/Os.
8. When commit 2 is issued, buffer B33 is rewritten, so the I/O is performed in series. Buffer B34 (the current buffer) is written out and the I/O to the two logs is performed in parallel. This commit request takes at least the time to do three I/O requests. When B34 is rewritten, the I/O is performed in series.

If the interval between the MQPUTs and the commits is very short compared to a disk I/O (for example less than 5 milliseconds), the following happens:

1. As before, when the second message is being put, and buffer B21 is about to be filled, because the number of full log buffers is greater than the value of WRTHRSH in CSQ6LOGP this signals the logger to write out pages up to (but not including) the current buffer. Buffers B1-B20 are written out, buffers B1-15 in one I/O, and buffers B16-B20 in a second I/O. The I/Os to each log data set is done in parallel.
2. If both the commits are issued while the above I/Os are happening, when the I/Os have finished, the logger writes buffers B21-B33 out in one I/O and buffer B34 (the current buffer) in a second I/O. The I/O for buffers B21-B33 is done in parallel, and the I/O for the current log buffer (B34) is also done in parallel to the two log data sets. The next time buffer B34 is rewritten, the I/O is done in series. The following table summarizes which buffers are written in each I/O:

<table>
<thead>
<tr>
<th>Long interval</th>
<th>Short interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1...B15 in parallel</td>
<td>B1...B15 in parallel</td>
</tr>
<tr>
<td>B16...B20 in parallel</td>
<td>B16...B20 in parallel</td>
</tr>
<tr>
<td>B21...B22 in parallel</td>
<td>B21...B33 in parallel</td>
</tr>
<tr>
<td>B23...B30 in parallel</td>
<td>B34 in parallel</td>
</tr>
<tr>
<td>B33 in parallel</td>
<td></td>
</tr>
<tr>
<td>B34 in parallel</td>
<td></td>
</tr>
<tr>
<td>Time taken: 8 I/O.</td>
<td>Time taken: 4 I/O. However, because more data is written in each I/O on average, each I/O takes longer than the long interval case</td>
</tr>
<tr>
<td>The next log check request rewrites B34 in series</td>
<td>The next log check request rewrites B34 in series.</td>
</tr>
</tbody>
</table>

The effect of one log I/O containing data from multiple transactions is called coat-tailing.

As a general rule, as the amount of data written to the log increases, the response time of requests requiring a log check increases.

In the example above, if the value of OUTBUFF was 80 (giving 20 log buffers) the put of message 2 would be suspended just before the write of buffer 21 because there are no free log buffers because buffers B1-B20 are all in use, with buffers B1-B15 being written to the log data sets. When the I/O completes and buffers B1-B15 are available again, the application can be resumed. The number of times that an application is suspended because no buffers are available is recorded in the log manager statistic QJSTWTB. If you get a nonzero value in this field, you should increase the value of OUTBUFF until the value of QJSTWTB remains at zero.

**MQPUT example**

| Table: Interpretation of the log statistics from the MQPUT and commit of 100 000-byte messages |
|-------------------------------------------------|-----------------|----------------|
| QJSTWWRNW       | Number of log writes with no wait | 215 |
| QJSTBFFL        | Number of log pages used | 2 550 |
| QJSTBFWR        | Number of calls to logging task | 200 |

- The information in the table is for 100 messages, so each message used about 25 log pages per message. Each log page is 4096 bytes long, so the 25 pages use 102 400 bytes. This includes the information about which pages have been changed, and information about the unit of work.
- For each MQPUT and commit there were two calls to the logging task, one call was made because the number of full log buffers was greater than the value of WRTHRSH (20), the other call was made during the commit.
- To write out 25 pages causes one I/O for 15 pages, another I/O for 9 pages, and an I/O for the current log buffer. The elapsed time taken to log the data is the duration of 4 I/Os, the parallel I/O for the 15 pages and the 9 pages, and two I/Os in series for the current log buffer.
**MQGET example**

| Table: Interpretation of the log statistics from the MQGET and commit of 100 000-byte messages |
|-------------------------------------------------|----------------------------------|
| QJSTWWRNW                                       | Number of log writes with no wait | 110 |
| QJSTBFFL                                        | Number of log pages used         | 29  |
| QJSTBFWR                                        | Number of calls to logging task  | 102 |

- The information in the table is for 100 messages so there is approximately one call to the logger per message.
- Only 29 pages were used to hold the log data. This shows that not very much data was logged and the same page used for several requests before the page was full.
- The same page was written out several times, even though it had not been completely filled.
- Because the current log buffer only was written each time, there was one I/O to each log, and because it was for the current buffer, these I/O were done in series.

**Interpretation of total time for requests**

In some measurements, the time taken to put a 100 000-byte message and a commit was 67 milliseconds on average, and the time to get a 100 000-byte message and a commit was 8 milliseconds on average. In both cases, most of the elapsed time was spent waiting for log I/O.

For the MQGET, the write I/Os to the dual logging devices were done in series. Because little data was written each time the connect time, when data was transmitted to the device, was small and RMF reports showed that the device had a short response time of 3-4 milliseconds. Two I/Os taking 3-4 milliseconds is close to the time of 8 milliseconds.

For the MQPUT, the write of the 15 and the 9 pages were done in parallel, and the write of the current buffer were done in series; in effect the time taken for four I/Os. Because a lot of data was written in a request, this caused a longer connect time, which leads to a longer overall DASD response time. RMF showed a response time of about 16-17 milliseconds for the log DASD. Four I/Os of 16-17 milliseconds is close to the measured time of 67 milliseconds.

**What is the maximum message rate for 100 000-byte messages?**

If we assume that:

- Put and commit of 100 messages use 2550 buffers (from the figures above)
- Get and commit of one message uses less than 1 buffer
- MQ writes a maximum 15 buffers for every I/O, where possible
- The I/O response time when writing 15 buffers per I/O is about 20 milliseconds
- The I/O response time for writing the current log buffer is 4 milliseconds
- There are no delays when writing to DASD (this includes within MVS and within the DASD subsystem)
- Concurrent copies of an application which puts a message, commits, gets the message, and commits again We can estimate the maximum message rate as follows:
  1. Out of the 2550 log buffers used for MQPUTs, 100 are written as the current log buffer, so 2450 can be written in parallel
  2. We can write up to 15 pages per I/O, so 2450 pages need 164 I/Os
  3. 164 I/Os, each taking 20 milliseconds gives a total of 3280 milliseconds
4. Each commit writes the current log buffer to each log data set in series. There are 100 commits for puts and 100 commits for gets. For two I/Os in series, each of 4 milliseconds, the total time for writing the current log buffers is \((100 + 100) \times 2 \times 4\) giving a total of 1600 milliseconds.

5. Total time for the I/O is 3280 + 1600 giving a total of 4880 milliseconds.

6. If it takes 4.88 seconds to process 100 messages, 20.5 messages could be processed in 1 second. This means that the theoretical absolute message rate is 20.5 messages per second.

This is the theoretical maximum with the given assumptions. In practice, the maximum will be different because the assumptions made are not entirely realistic. In a measurement made using a requester/reply application model where a CICS transaction put a 100 000-byte message to a server, and received the same message back, the transaction rate was 10-11 transactions (21 messages) per second.

END OF DOCUMENT