Redpaper

Graham Rawson

IBM CICS Performance Series: CICS TS V4.2 and Java Performance

This IBM® Redpaper™ publication reviews performance-related changes introduced in IBM CICS® TS V4.2 that support Java. The paper describes in particular the use of multi-threaded JVM server environments and the support of 64-bit Java 6.0.1 JVMs. This paper is intended for CICS systems architects, administrators, and performance analysts who want to plan, deploy, and monitor these new Java environments.

This paper is one of a series focused on CICS performance written by members of the IBM Hursley CICS development community. The subject matter in this series is based on feedback from CICS customers.

What this paper contains

This paper contains a review of CICS TS V4.2 Java support illustrated with results from case studies. The paper covers the following topics:

- History of Java support in CICS
- New features of CICS TS V4.2
- Comparing CICS TS V4.2 and CICS TS V4.1, showing benefits of the new multi-threaded Java virtual machine (JVM) configuration and 64-bit Java 6.0.1
- Choosing between pooled JVMs and JVM server
- Using shared classes
- Sizing the Java heap
- Choosing JVM garbage collection (GC) policies
- Paying for a JVM run in CICS
- Using 64-bit compressed references
- Tuning IBM Language Environment® supporting the CICS/Java environment
- Using IBM zSeries® special purpose processors, IBM System z® Application Assist Processor (zAAP) and IBM System z Integrated Information Processor (zIIP)
- Tools for investigating Java performance in CICS

History of Java support in CICS

The following releases of CICS TS supported a number of products and offered a range of features:

- CICS TS V1.3
  - Supported Java software development kit (SDK) V1.1.
  - Started applications either as Java Program Objects or as Java bytecodes within a JVM.
    - Objects were native code executables created by the High Performance Java (HPJ) product by compiling Java bytecodes.
    - Startup of an HPJ environment was costly, and thus a technique to reuse the HPJ environment was developed, called *hot pooling*.
    - JVM support in CICS TS V1.3 was costly because each call of a CICS Java program required a JVM to be created, used, and then terminated.
- CICS TS V2.1
  - Supported Java SDK V1.3.
  - Supported HPJ for migration to JVM.
  - Supported a pool of JVMs in a resettable mode, providing a reusable JVM that reset storage between uses. CICS TS V2.1 also made it possible to share Java classes and just-in-time (JIT) compiled code.

- CICS TS V2.2 and V2.3
  - Supported Java SDK V1.4.1.
  - Supported HPJ for migration to JVM.
  - Ran pooled JVMs in resettable, single, or continuous modes.
    - Resettable mode, as described for CICS TS V2.1, provided a reusable JVM that reset storage between uses.
    - Single-use mode created and deleted a JVM with every invocation of a CICS Java program and thus was costly.
    - Continuous mode, introduced in CICS TS V2.3, provided a reusable JVM, like resettable mode, but without the processor resource cost of resetting storage between uses.
- CICS TS V3.1
  - Supported Java SDK V1.4.2.
  - Supported pooled JVMs in resettable, single, or continuous modes.
  - Supported a shared class cache owned by a Master JVM.
  - Did not support HPJ.
- CICS TS V3.2
  - Supported Java SDK V1.4.2 and SDK V5.0 (31-bit only).
  - Supported pooled JVMs in single or continuous modes.
  - Supported shared class caches for Java SDK V1.4.2 or V5.0.
- CICS TS V4.1
  - Supported Java SDK V6.0 and V6.0.1 (31-bit only).
  - Required PTFs UK71181 and UK71182 to support Java V6.0.1.
  - Supported pooled JVMs in single or continuous modes.
  - Implemented multi-threaded JVM server infrastructure for CICS system processing only, such as dynamic script support.
- CICS TS V4.2
  - Supported Java V6.0.1 (64-bit only).
  - Supported pooled JVMs in single or continuous modes.
  - Supported multi-threaded JVM server for general use with applications deployed as OSGi bundles.
  - Supported Open Services Gateway Initiative (OSGi), optimizing operational control and making it possible for new versions of the applications to be introduced without restarting the JVM.

For instructions on upgrading from CICS TS V3.1, V3.2, or V4.1 to CICS TS V4.2, see the CICS Information Center:

https://publib.boulder.ibm.com/infocenter/cicsts/v4r2/index.jsp

New features of CICS TS V4.2

In this section, we look at what is new in CICS TS V4.2 for CICS TS Java performance.

Support for 64-bit Java V6.0.1

CICS TS V4.2 uses the IBM 64-bit SDK for z/OS®, Java Technology Edition, Version V6.0.1, in both pooled JVMs and JVM server configurations. No support is provided for 31-bit JVMs with CICS TS V4.2. Java V6.0.1 includes enhancements to JIT optimization and use of z196 hardware with new hardware instructions and out-of-order pipeline execution. Java V6.0.1 is not distributed with CICS TS V4.2 but is available at no charge from the following site:

http://www.ibm.com/systems/z/os/zos/tools/java/

Using a 64-bit JVM provides relief for virtual storage constraints by moving most of the memory used by both pooled JVMs and JVM server above the 64-bit bar, as shown in Figure 1. Moving memory makes it possible for a single CICS TS V4.2 region to support many more JVMs than earlier versions of CICS, and each JVM can have much larger heaps. With CICS TS V4.2, it is possible to host traditional and Java workloads in a single CICS TS V4.2 region without encountering "short-on-storage" conditions.

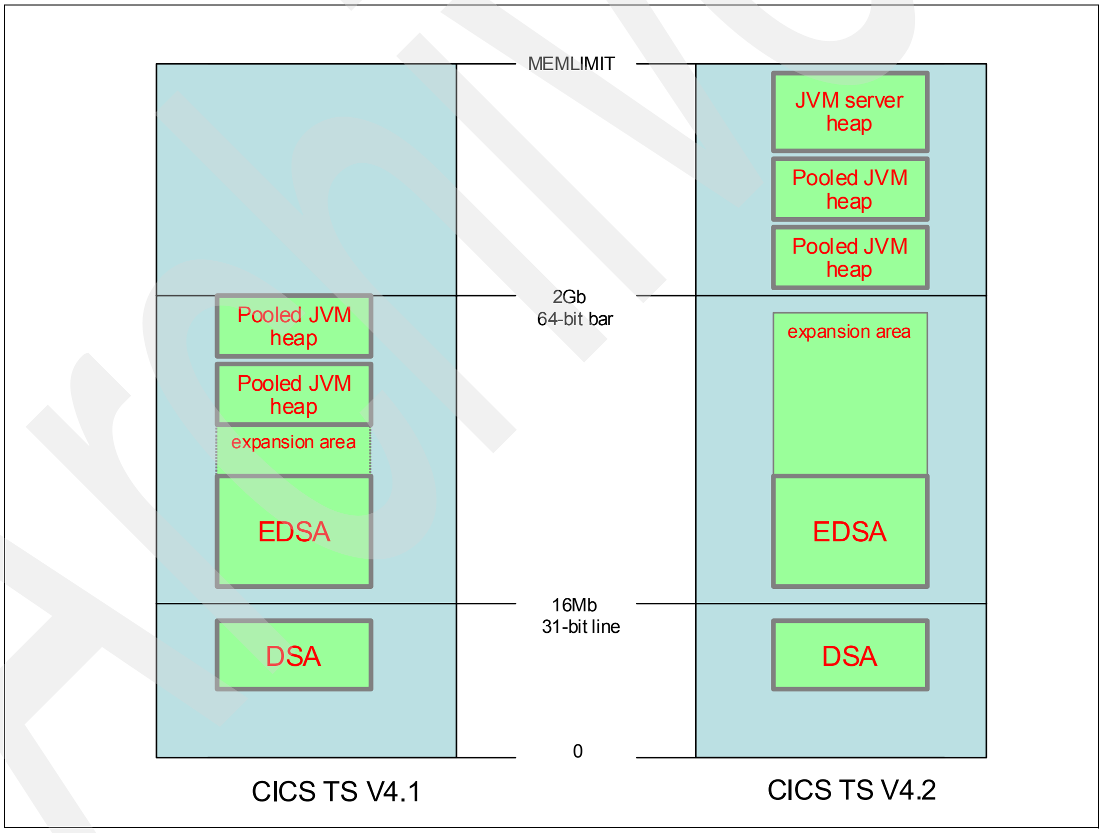


Figure 1 Increasing available 31-bit memory space

A potential disadvantage of using a 64-bit JVM is the associated increase in the size of Java objects needed to accommodate 64-bit object references. Usually, Java applications using 31-bit memory with CICS TS V4.1 and Java V6.0 use a larger amount of 64-bit memory if you upgrade to CICS TS V4.2 and Java V6.0.1. However, you can moderate this increase by using the *compressed references* feature of the IBM JVM, which reduces the size of 64-bit Java objects. Compressed references are described in more detail in “Using 64-bit compressed references” on page 21.

Pooled JVMs and JVM server configurations

Each JVM within CICS, either pooled JVM or a JVM server, runs as a z/OS UNIX process. A Language Environment enclave provides the UNIX execution environment for such processes, managing the UNIX programs and memory. You can find more details about tuning this Language Environment enclave in “Tuning the IBM Language Environment” on page 23.

Before CICS TS V4.2, each CICS Java program ran in a reusable pooled JVM that supported only single-threaded use. Although pooled JVMs share some attributes, each has its own Java heap, Java class libraries, and code compiled by JIT, as shown in Figure 2.

Each pooled JVM thread in Figure 2 runs as a separate IBM MVS™ task with its own associated MVS Task Control Block (TCB) managed by CICS. The maximum number of such tasks a CICS region supports is specified with the CICS system parameter **`MAXJVMTCBS`**.

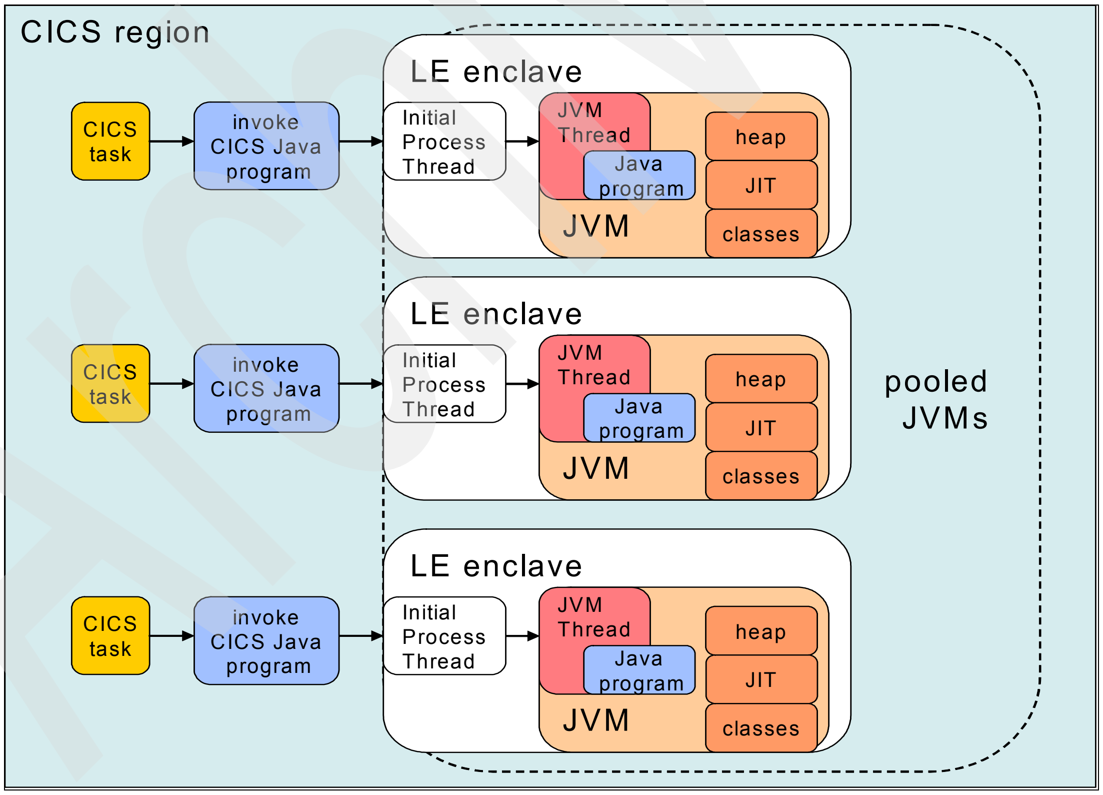


Figure 2 Language Environment enclave structure for pooled JVMs

CICS TS V4.2 introduces the JVM server configuration for general use. Using the JVM server, you can run multiple concurrent CICS tasks in a single multi-threaded JVM. These multiple requests share the Java heap, class libraries, and code compiled by JIT. The JVM server provides the potential benefit of requiring less memory to run a workload because concurrent users share heap, JIT code, and Java classes, as shown in Figure 3.

Each JVM server thread in Figure 3 runs as a separate MVS task with its own associated TCB managed by CICS. The maximum number of such tasks a JVM server supports is specified by the **THREADLIMIT** parameter associated with the JVM server.

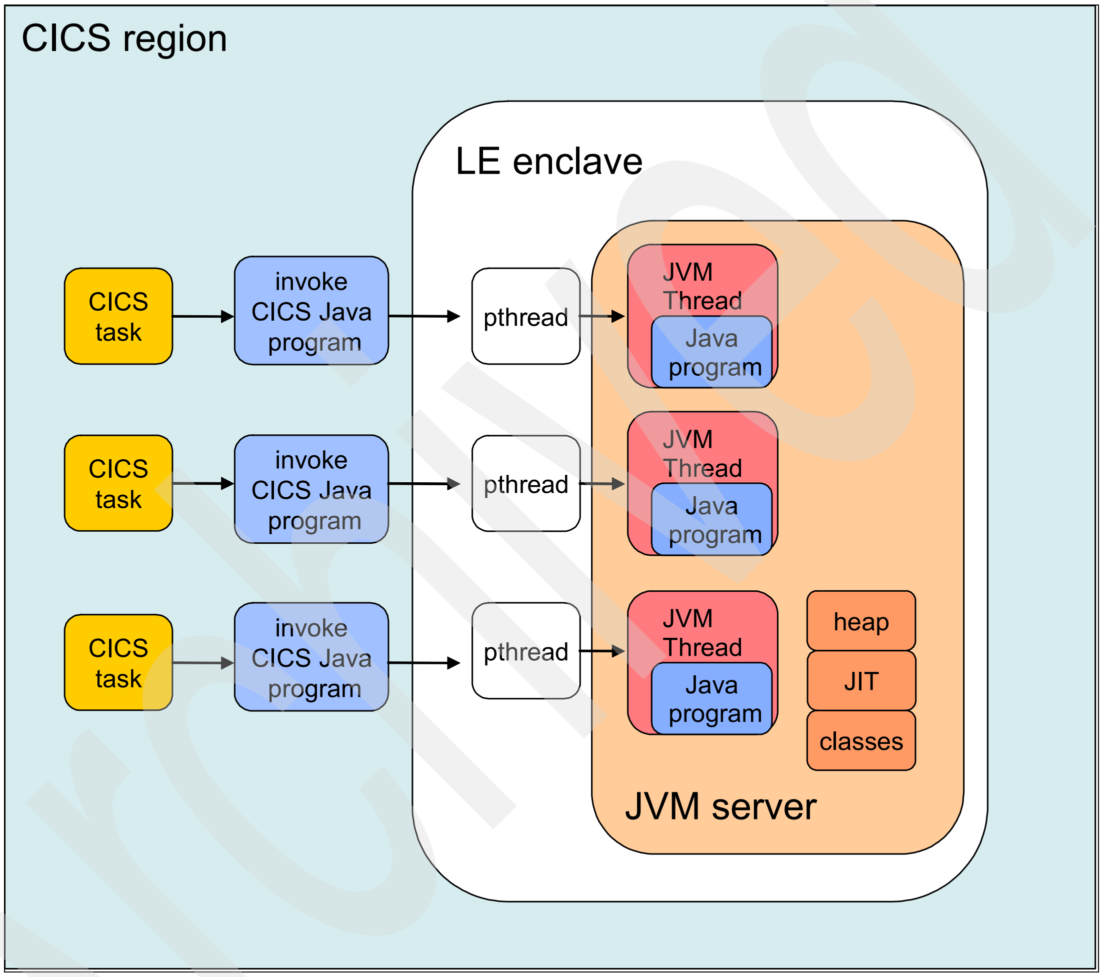


Figure 3 Language Environment enclave structure for JVM server

Changes in JVM garbage collection

IBM Java V6.0.1 supports a number of garbage collection (GC) policies. These GC policies can be specified in the JVM profile used by both pooled JVM and the JVM servers in both CICS TS V4.1 and V4.2. However, the default policies and settings supplied by CICS differ between CICS TS V4.1 and V4.2.

CICS TS V4.1 supports Java V6.0, which has a default **`gcpolicy`** of `optthruput`. Using a **`gcpolicy`** setting of `optthruput` means that the memory management functions of the JVM treat the Java heap as a single space. The JVM optimizes this policy to deliver high throughput, but it can occasional produce long pauses. CICS TS V4.1 usually avoids these long pauses by forcing GC using CJGC, a transaction supplied by CICS, in each pooled JVM when it detects that the Java heap reaches a heap occupancy exceeding **`GC_HEAP_THRESHOLD`**. If the **`GC_HEAP_THRESHOLD`** parameter value is not specified, a default value of 85% is used.

CICS TS V4.2 supports Java V6.0.1, which has a default policy of `gencon`. When you use a **`gcpolicy`** setting of `gencon`, the memory management functions of the JVM treat the Java heap as separate generational areas. This policy is optimized to deliver lower pause times while maintaining high throughput. When a JVM server supports concurrent threads, it forces the pooled JVM to use the GC mechanism. Using this mechanism is undesirable because the forced GC can interrupt other CICS tasks that are still running Java applications. The JVM manages GC in a JVM server whenever a GC event occurs (such as an allocation failure). You do not see any CJGC transactions running in a JVM server.

The *Diagnostics Guide* for IBM SDK Java contains a comprehensive explanation of how the JVM manages memory and garbage collection. The publication is available at this site:

`http://www.ibm.com/developerworks/java/jdk/diagnosis/`

Comparing CICS TS V4.2 and CICS TS V4.1

In the following section, we compare CICS TS V4.2 with CICS TS V4.1 in the following areas:

- Throughput
- Transaction costs
- Response times
- z/OS 31-bit memory use
- z/OS 64-bit memory use

Comparing throughput

We collected the data shown in this section using a workload of CPU-intensive Java applications. These applications included a mixture of CICS application programming interface (API) calls using the JCICS Java class library. JCICS is the Java equivalent of the EXEC CICS API that you use with other languages supported by CICS.

We maintained a high transaction injection rate to drive the host system to its maximum use. The host system was an LPAR of an IBM z196 system 2817 model M80-779 with eight dedicated processors and 16 Gb of memory. At 5-minute intervals, we increased the available threads. If workload execution began with a single pooled JVM, we incremented the system **`MAXJVMTCBS`** count to increase these threads. If workload execution began with the JVM server, we incremented the **`THREADLIMIT`** count associated with the JVM server to increase the threads.

Figure 4 shows the transaction rate per second for the final minute of each 5-minute measurement interval. We chose the last minute to minimize the inclusion of any startup costs associated with a new pooled JVM. Because the workload comprises processor-intensive applications, each thread can drive a single processor to high use. For both the pooled JVM and JVM server configurations, using eight threads makes it possible for the host z/OS system to use 99.9% of the available processor resources.

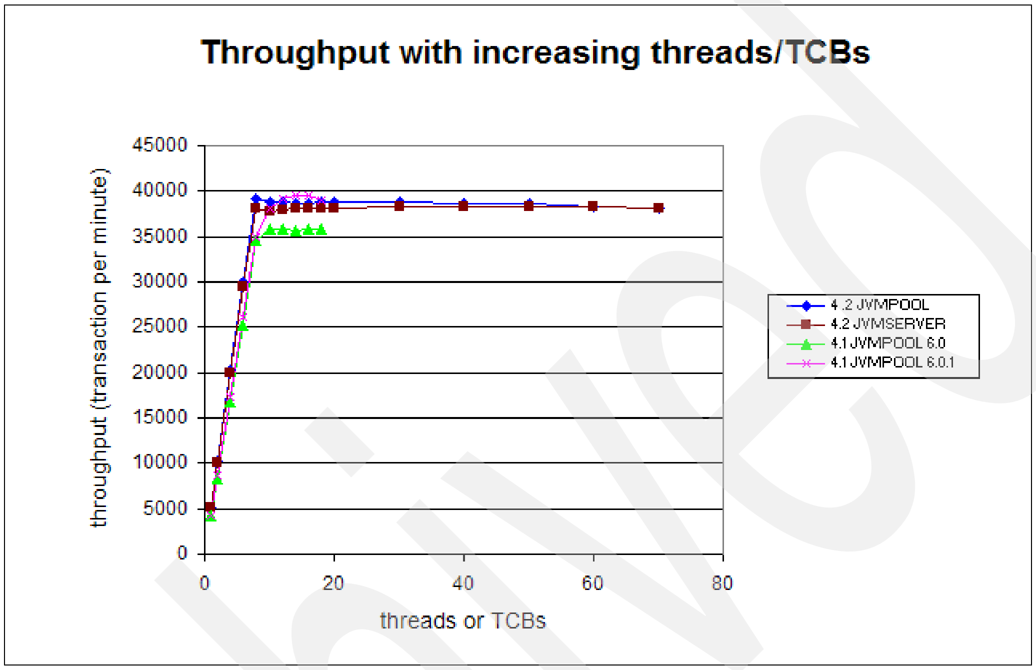


Figure 4 Comparing throughput with increasing numbers of threads or TCBs

The improved performance characteristics of Java V6.0.1 produce a higher throughput for CICS TS V4.2 using either pooled JVMs or a JVM server, as opposed to CICS TS V4.1 with Java V6.0. When used with CICS TS V4.1, Java V6.0.1 also delivers improved throughput compared to CICS TS V4.1 with Java V6.0. The CICS TS V4.1 regions use a 31-bit JVM and cannot support more than 20 pooled JVMs before using all the available 31-bit memory in the address space. The pooled JVMs in CICS TS V4.2 are not constrained in this manner, and thus CICS TS V4.2 can start many more pooled JVMs than CICS TS V4.1.

With CICS TS V4.2, the multi-pooled JVM environment produces slightly higher throughput than a single multi-threaded JVM server. However, this higher value most likely results from the additional functions provided by JVM server, such as Open Services Gateway initiative (OSGi) support and multiple thread management. The 31-bit Java V6.0.1 with CICS TS V4.1 provides lower throughput with smaller numbers of TCBs than CICS TS V4.2. However, at the highest throughput levels, 31-bit Java V6.0.1 delivers the best performance with the applications used in this study. Figure 5 illustrates this performance and plots lower thread/TCB counts than in Figure 4 on page 8.

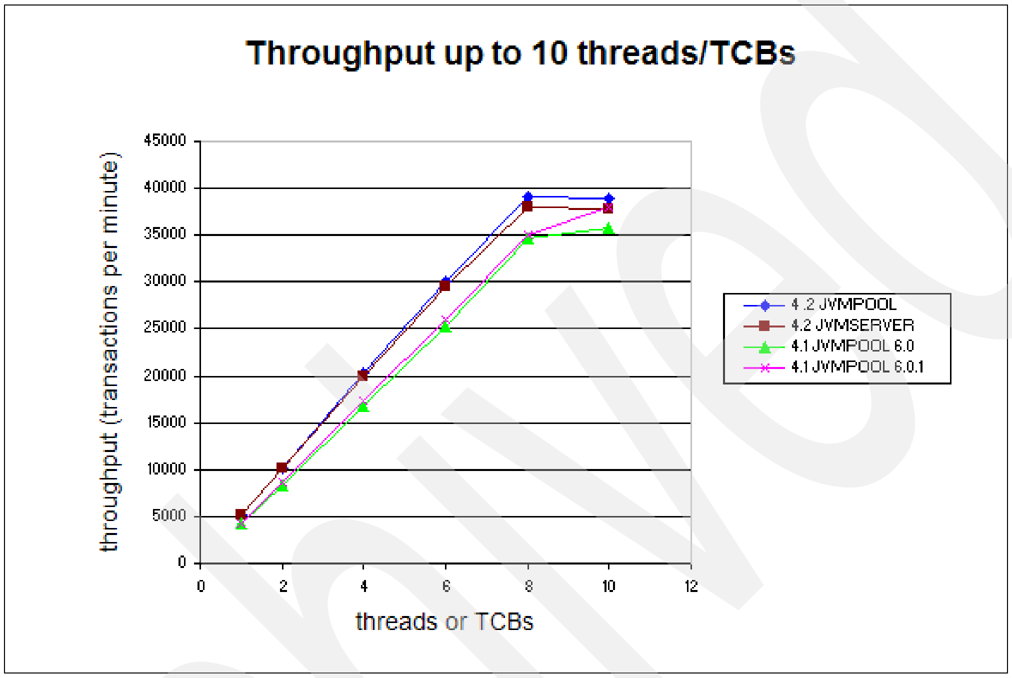


Figure 5 Comparing throughput with increasing low numbers of threads or TCBs

Figure 6 shows how well both CICS TS V4.1 and CICS TS V4.2 scale with increasing workload. We measured total system use for increasing transaction rates with 16 standard processors and one zIIP online. We found that the CICS TS V4.2 region can successfully drive the host z/OS system to high processor use. However, a single CICS TS V4.1 region cannot support sufficient pooled JVMs to fully use the host z/OS system. Completely using all the processor resources requires an additional CICS TS V4.1 region.

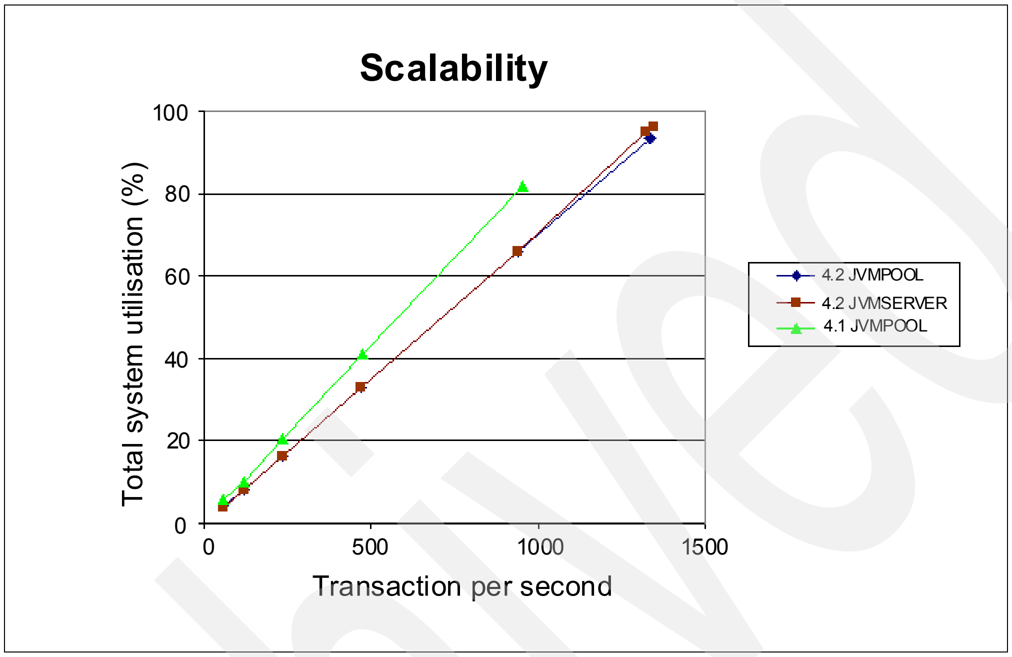


Figure 6 Comparing system use with increasing workload

Comparing transaction costs

We collected the data in Figure 7 using the same workload of processor-intensive Java applications used in the throughput measurements in Figure 4 on page 8. We used eight dedicated processors of an IBM z196 system 2817 model M80-779. The pooled JVM configuration set **`MAXJVMTCBS`** to `20`, the maximum number of JVMs that this region can support.

The JVM server set **`THREADLIMIT`** to `40` to ensure that there were sufficient threads available to handle peak demand, minimizing the JVM server thread wait time. This wait time is reported in the 'elapse time' fields of JVM server section of a CICS statistics report, as shown in the example in Figure 22 on page 27. We increased the workload injection rate at 10-minute intervals. We determined processor cost per transaction using the User processor time data field (USRCPUT) reported by CICS monitoring.

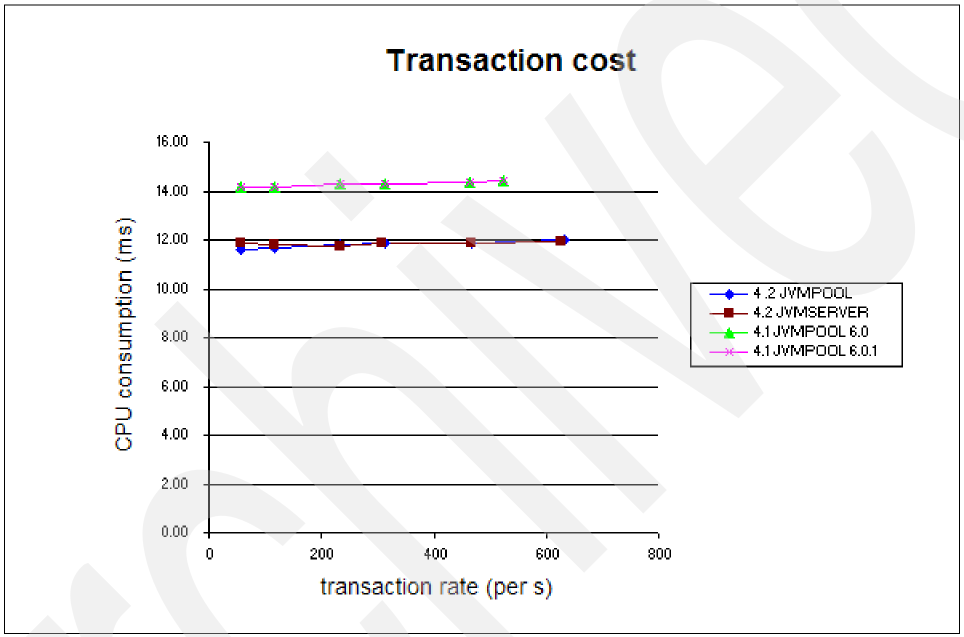


Figure 7 Comparing transaction costs with increasing workload

In this configuration, CICS TS V4.2 with Java V6.0.1 delivers a lower transaction cost than CICS TS V4.1. This behavior is also reflected at the lower thread numbers and throughput rates of CICS TS V4.2, as shown in Figure 5 on page 9.

Comparing response times

We extracted the data in Figure 8 from transaction response time data reported by CICS monitoring. We measured three different types of application: a HelloWorld supplied by CICS, mathematical functions performed on randomly generated integers, and a bubble sort of 5000 randomly created integers. The benefits of lower response times result from a range of improvements provided by Java V6.0.1.

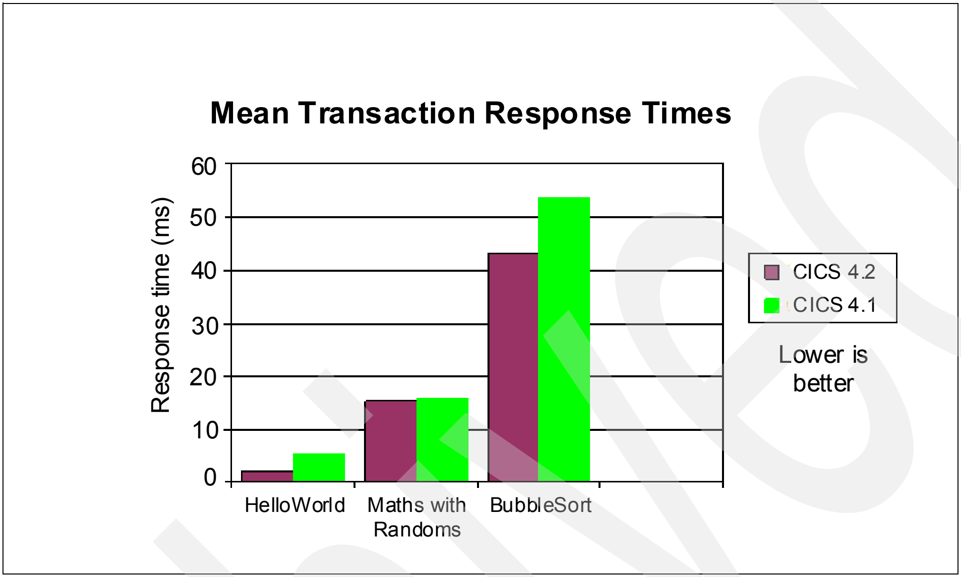


Figure 8 Comparing response times for different types of transaction

Comparing z/OS 31-bit memory use

Each pooled JVM or JVM server runs in its own Language Environment enclave with its own 31-bit and 64-bit memory areas. The 64-bit JVM that both the pooled JVM and JVM server use in CICS TS V4.2 still has a small requirement for some 31-bit memory. However, most of the storage used by the JVM, such as the Java heap, is in 64-bit memory. The 31-bit JVM used by CICS TS V4.1 must allocate memory from the same area of 31-bit memory used to allocate CICS extended dynamic storage areas (EDSAs). This competition for the limited 31-bit memory space in a CICS region restricts the number of JVMs that a CICS TS V4.1 region can support.

This limit effectively disappears in CICS TS V4.2. To illustrate this change, Figure 9 shows the amount of 31-bit memory used by a CICS region as reported in the IBM Resource Measurement Facility™ (IBM RMF™) Virtual Storage Activity report (VSTOR). For the measurements in Figure 9, we configured each CICS TS V4.1 pooled JVM with a fixed heap of 20 MB and each CICS TS V4.2 pooled JVM with a fixed heap of 30 MB. We configured the single CICS TS V4.2 JVM server with a fixed heap of 300 MB. We fixed the JVM heap sizes by setting the JVM initialization parameters **`-Xms`** and **`-Xmx`** to the same values, in this case, `-Xmx300M and -Xms300M`.

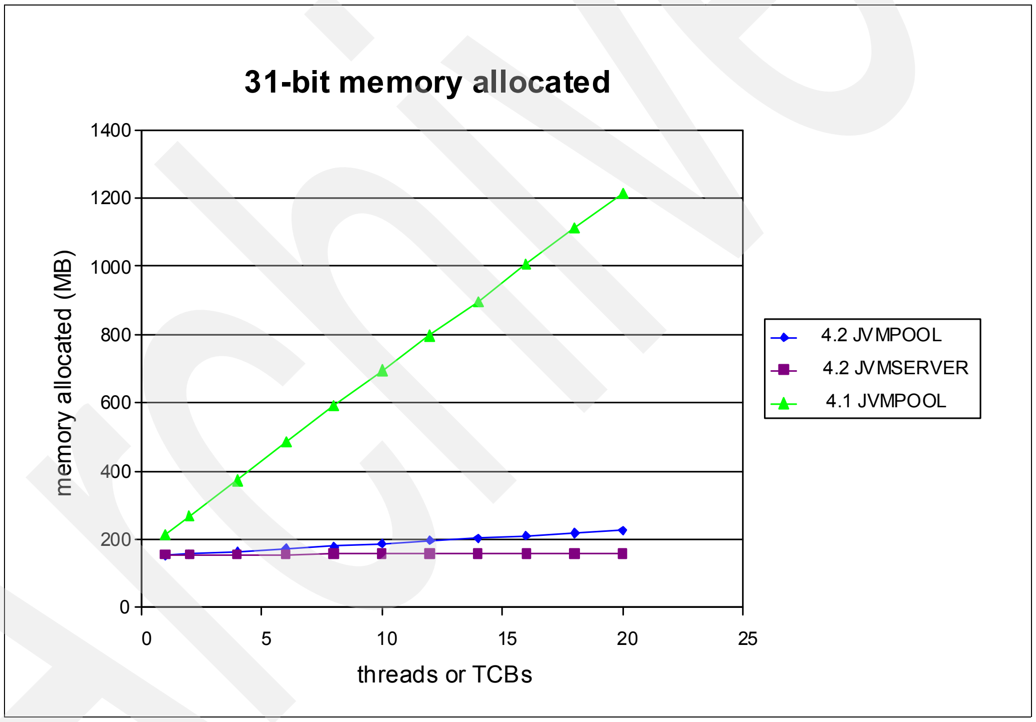


Figure 9 Comparing 31-bit memory usage

Each pooled JVM in the CICS TS V4.1 configuration uses about 50 MB of 31-bit memory. The RMF data shows that the CICS TS V4.1 region grows from about 200 MB to about 1200 MB when the CIS region uses 20 JVMPOOL JVMs concurrently effectively using all the available 31-bit memory in the CICS region. With CICS TS V4.2, each JVM uses about 3 MB of 31-bit memory. With 20 pooled JVMs, the CICS TS V4.2 region grows from about 150 MB to about 210 MB. The JVM server in CICS TS V4.2 shows no measurable increase in memory usage when increasing the numbers of used threads.

Comparing z/OS 64-bit memory use

A 64-bit JVM used by CICS TS V4.2 uses more memory than its 31-bit counterpart in CICS TS V4.1, but this memory comes from the less constrained 64-bit area. The MEMLIMIT value associated with the CICS address space restricts the amount of 64-bit memory available to CICS. The z/OS memory management system manages the 64-bit area as memory objects and RMF reports its use. To be available for processing, z/OS memory management must allocate any memory (24-bit, 31-bit, or 64-bit) to *real memory*. Real memory is the physical memory belonging to a computer system. Real memory use is likely to become the restricting factor affecting 64-bit memory use because excessive demand for real memory causes the z/OS system to page, degrading overall system performance.

A concise description of virtual, real, and auxiliary storage in z/OS is available from the SHARE group:

http://share.confex.com/share/115/webprogram/Session7511.html

Storage Manager statistics in CICS TS V4.2 provide a measure of how many real memory frames are used to support 64-bit memory objects by a CICS region. Figure 10 shows the real memory used by CICS TS V4.2 regions using pooled JVMs and the JVM server to provide increasing numbers of TCBs or threads. In this measurement, each pooled JVM has a fixed Java heap of 30 MB, and the JVM server has a fixed Java heap of 300 MB. During these measurements, there was no constraint on real memory.

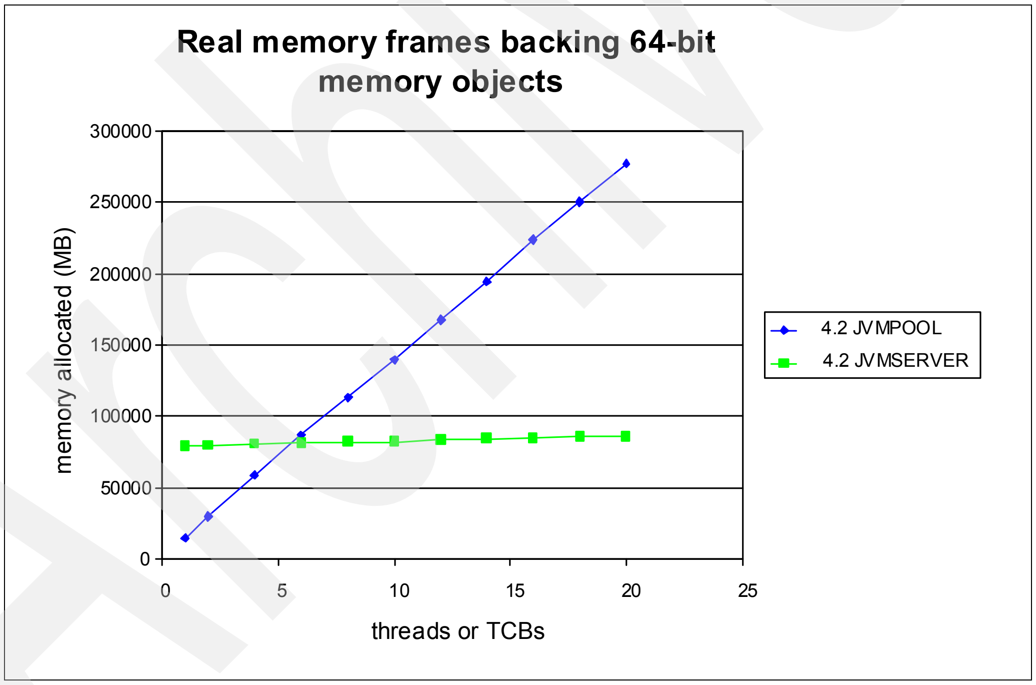


Figure 10 Comparing real memory used to support 64-bit memory objects

Each pooled JVM uses about 14,000 real memory frames. In this case, the page size is 4 KB, and each pooled JVM uses about 53.4 MB of real memory. The JVM server uses about 78,000 memory frames with an additional 330 memory frames for every additional thread used. Thus, a JVM server using 20 concurrent threads uses about 336 MB of real memory.

Choosing between pooled JVMs and JVM server

In the throughput data shown in Figure 4 on page 8, we found that CICS TS V4.2 using multiple pooled JVMs gives slightly better performance in terms of processor cost per transaction than a single multi-threaded JVM server. The JVM server does introduce a small amount of processor use to provide an OSGi framework and to manage multiple threads. This amount of processor use is accentuated by using the simple applications in this workload. A more complex workload using many Java classes makes better use of a JVM server environment because it can use the larger heap and single copy of JIT and JIT methods provided by the JVM server.

The main benefit of using a JVM server with this workload is that it delivers similar throughput at similar processor cost but uses less memory than running multiple JVMPOOL JVMs.

The Announcement Letter for CICS TS V4.2 contains, as a statement of direction, that IBM intends a future release to discontinue support for pooled JVMs and encourages customers to upgrade to a JVM server configuration:

`http://www.ibm.com/common/ssi/cgi-bin/ssialias?infotype=an&subtype=ca&supplier=897&appname=IBMLinkRedirect&letternum=ENUS211-080`

Using shared classes

The JVM server directly uses the shared class support provided by Java V6.0.1. Unlike pooled JVMs, no management interface (using **`SPI`** or **`CEMT`** commands, for example) exists between CICS and the shared class cache. Class data sharing between JVMs is described in the *IBM User Guide for Java V6 on z/OS* in the IBM SDK and Runtime Environment Java Technology Edition Version 6 Information Center:

`http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp`

The ability to reuse bootstrap and application classes and store the AheadOfTime compiled methods in the shared class cache delivers these benefits:

- Faster JVM start
- Faster application start
- Reduced memory consumption if using multiple JVMs

These benefits are of most value in configurations that use multiple JVMs and support Java applications with large numbers of classes. However, you can see startup improvements even in a CICS region with a single JVM server supporting simple Java applications. Table 1 shows the processor costs in starting a JVM server in a single CICS TS V4.2 region. The table shows the costs first without using a shared class cache, then with an empty shared class cache loaded on first use, and finally, using the previously loaded cache.

The cost is reported as the processor time used by each CJSR transaction supplied by CICS and recorded in the USRCPUT field of the CICS monitoring data for that transaction. The CJSR transaction is responsible for starting the JVM server.

Table 1 Comparing JVM startup costs with shared class cache

	Processor time used by CJSR transaction (ms)
No shared class cache	37.616
Shared class cache first use	39.913
Shared class cache second use	31.825

These results demonstrate that first loading the shared class cache requires extra processor resources but later uses less of these resources to initialize the JVM server.

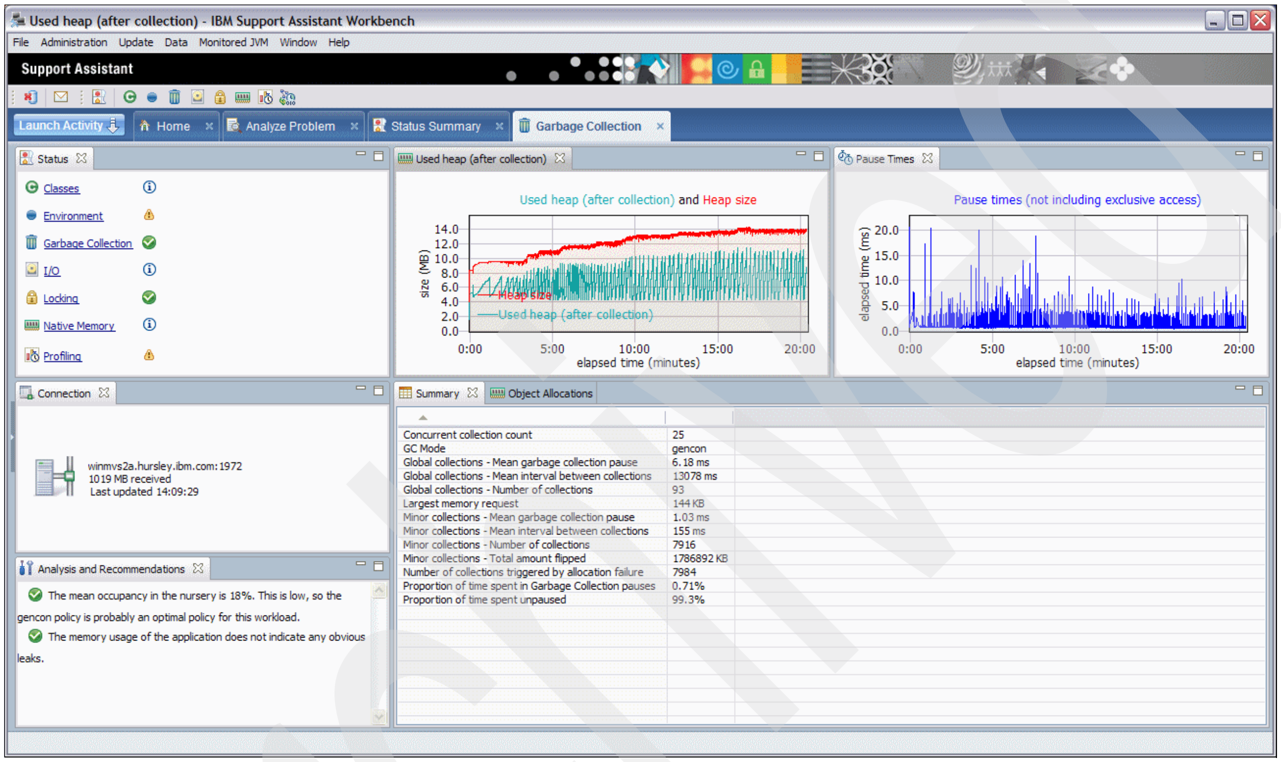
Sizing the Java heap

The Health Center and Garbage Collection and Memory Analyzer tools are included in the IBM Support Center package (described in "Tools for investigating Java performance in CICS" on page 26). This tool shows the amount of Java heap used and the time taken to manage the contents of the heap. JVM server statistics in CICS TS V4.2 are extended to include Java heap data. The goal in sizing the Java heap is to minimize the heap size while maintaining an acceptable impact by GC pause times on the application response times.

If you are upgrading from a CICS TS V4.1 pooled JVM configuration, then consider increasing any heap sizes you have by about 50% to allow for the increased object sizes in the 64-bit JVM. You can eradicate much of this increase by using the Compressed References mode described in "Using 64-bit compressed references" on page 21. Even for a pooled JVM migration, it is advisable to check the heap sizings using Java tools.

To size the heap for a new application set or for a new JVM server configuration, we suggest starting with the default heap settings. Collect and analyze GC data using Java tools, and use these results to set more suitable heap sizes. The JVM server statistics provide a cheap and easy way of ensuring that the GC behavior is maintained in terms of pause times and the rate of garbage creation. The JVM can expand and contract the Java heap according to demand, but this dynamic resizing creates a small processor usage cost. Generally, you can gain best performance from fixed heap sizes, but a fixed size heap might not be suitable for workloads that vary largely in their demands for heap storage. This section illustrates some heap tuning activities.

A workload increases in transaction frequency, reaching a steady state after about 5 minutes. Using the default heap settings, the output from the Health Center (Figure 11) shows that the JVM automatically allocated the initial heap size at 4 MB but steadily increased this size to accommodate increasing application demand for heap storage. The heap size reached a steady state of about 14 MB after about 16 minutes. Although rising application demand triggered many minor collections (approximately 400 per minute), pause times account for only 0.7% of the elapsed time. A value of less than 1% is good. If this value exceeds 13%, the default behavior of the JVM is to expand the Java heap. (This value can be modified using the **`-Xmaxt`** JVM parameter.)


Used heap (after collection) - IBM Support Assistant Workbench
File Administration Update Data Monitored JVM Window Help
Support Assistant
Launch Activity
Home
Analyze Problem
Status Summary
Garbage Collection
Status
Classes
Environment
Garbage Collection
I/O
Locking
Native Memory
Profiling
Used heap (after collection)
Used heap (after collection) and Heap size
size (MB)
Heap size
Used heap (after collection)
elapsed time (minutes)
Pause Times
Pause times (not including exclusive access)
elapsed time (ms)
elapsed time (minutes)
Connection
Analysis and Recommendations
Summary
Object Allocations

winmvs2a.hursley.ibm.com:1972
1019 MB received
Last updated 14:09:29

✓ The mean occupancy in the nursery is 18%. This is low, so the gencon policy is probably an optimal policy for this workload.

✓ The memory usage of the application does not indicate any obvious leaks.

Concurrent collection count	25
GC Mode	gencon
Global collections - Mean garbage collection pause	6.18 ms
Global collections - Mean interval between collections	13078 ms
Global collections - Number of collections	93
Largest memory request	144 KB
Minor collections - Mean garbage collection pause	1.03 ms
Minor collections - Mean interval between collections	155 ms
Minor collections - Number of collections	7916
Minor collections - Total amount flipped	1786892 KB
Number of collections triggered by allocation failure	7984
Proportion of time spent in Garbage Collection pauses	0.71%
Proportion of time spent unpaused	99.3%

Figure 11 Health Center output for default heap

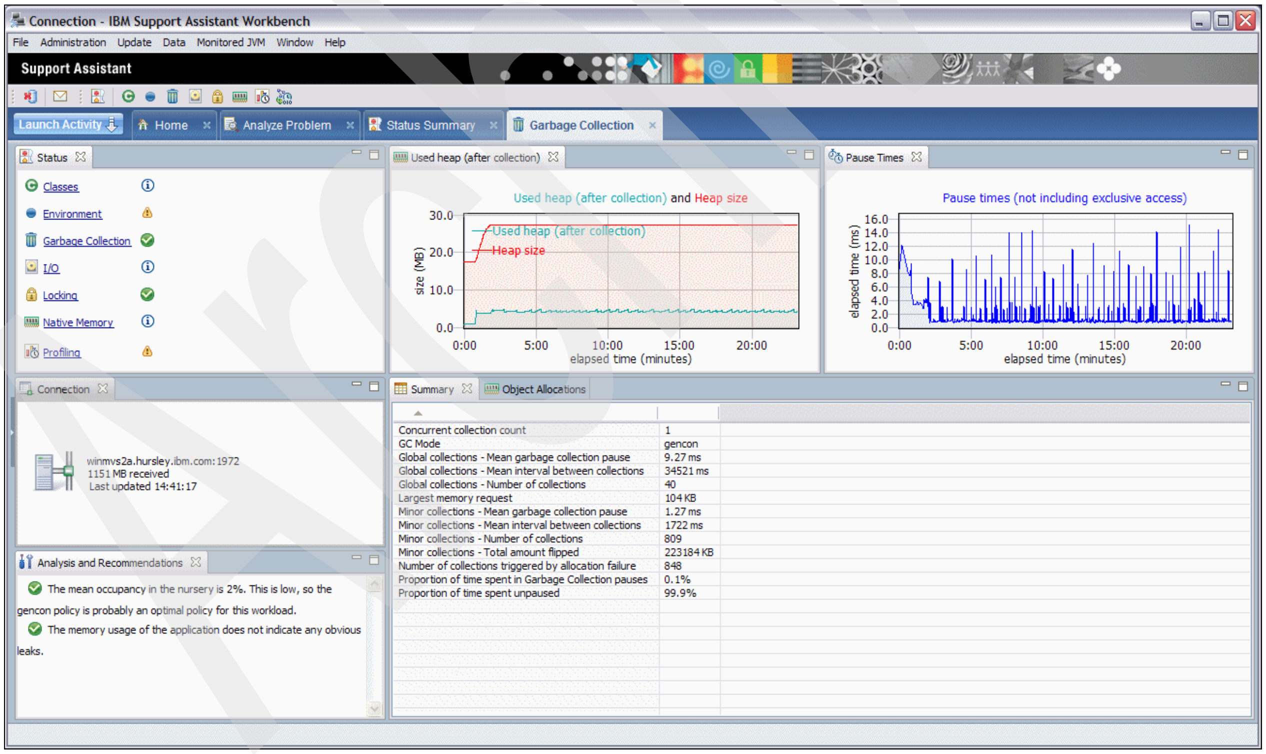
Using the default `gencon` GC policy splits the Java heap into two areas, the new or *nursery* area, and the old or *tenured* area. CICS JVM server statistics call new or nursery activity *minor* GC and call old or tenured activity *major* GC.

The CICS JVM server interval statistics collected for the final minute of this workload reflect the used peak heap size of about 15 MB, as shown in Figure 12. The statistics also reflect pause times of about 0.74% (113 ms major GC + 355 ms minor GC within the 60-second interval). The CICS utility program **DFHSTUPT** produces this data by processing CICS statistics System Management Facilities (SMF) records.

```
JVM server JVM creation time. . . . :  11/09/2011 13:48:44.8215
JVM server current heap size . . . .:    6856392
JVM server initial heap size . . . .:         4M
JVM server maximum heap size . . . .:       512M
JVM server peak heap size . . . . . :  15259960
JVM server heap occupancy . . . . . :    5266032
JVM server Garbage Collection policy:  -Xgcpolicy:gencon
JVM server major GC collections. . .:          4
JVM server elapsed time in major GC.:        113
JVM server major GC heap freed . . .:  34568720
JVM server minor GC collections. . .:        411
JVM server elapsed time in minor GC.:        355
JVM server minor GC heap freed . . .:      1086M
```

Figure 12   One minute of JVM server statistics for default heap

Setting the nursery to 25 MB and the tenured area to 5 MB (a total of 30 MB) produces fewer minor collections, as shown in Figure 13.



winmvs2a.hursley.ibm.com:1972
1151 MB received
Last updated 14:41:17

✓ The mean occupancy in the nursery is 2%. This is low, so the gencon policy is probably an optimal policy for this workload.

✓ The memory usage of the application does not indicate any obvious leaks.

Concurrent collection count	1
GC Mode	gencon
Global collections - Mean garbage collection pause	9.27 ms
Global collections - Mean interval between collections	34521 ms
Global collections - Number of collections	40
Largest memory request	104 KB
Minor collections - Mean garbage collection pause	1.27 ms
Minor collections - Mean interval between collections	1722 ms
Minor collections - Number of collections	809
Minor collections - Total amount flipped	223184 KB
Number of collections triggered by allocation failure	848
Proportion of time spent in Garbage Collection pauses	0.1%
Proportion of time spent unpaused	99.9%

Figure 13   Health Center output for fixed heap

The accompanying CICS JVM statistics report (Figure 14) shows a reduced pause time of 0.1% (20 ms major GC + 47 ms minor GC within the 60-second interval).

The statistics report also shows the reduced number of collections and small pause times.

```
JVM server JVM creation time. . . . :  11/09/2011 14:17:54.0701
JVM server current heap size . . . .:  22982304
JVM server initial heap size . . . .:       30M
JVM server maximum heap size . . . .:       30M
JVM server peak heap size . . . . . :    30625K
JVM server heap occupancy . . . . . :   5126408
JVM server Garbage Collection policy:  -Xgcpolicy:gencon
JVM server major GC collections. . .:         2
JVM server elapsed time in major GC.:        20
JVM server major GC heap freed . . .:   6704792
JVM server minor GC collections. . .:        39
JVM server elapsed time in minor GC.:        47
JVM server minor GC heap freed . . .:      956M
```

Figure 14 One minute of JVM server statistics for fixed heap

Choosing JVM garbage collection policies

The default GC policy for both Java V6.0.1 and CICS TS V4.2 is `gencon`, but you are at liberty to use any of the GC policies and tuning parameters that are detailed in the *Java Diagnostics Guide*.

The default GC policy for the previous versions of Java and CICS was `optthruput`. The `optthruput` policy treats the Java heap as a single space. However, `gencon` splits the heap into an area to allocate new objects (a nursery space) and an area to keep long-lived objects (a tenure space). Garbage collection of a single space heap is normally a less frequent, but longer-lasting, process than garbage collecting in a smaller nursery area. The `gencon` policy is better suited to Java workloads that create many small short-lived objects or large long-lived objects, typical behavior of a transactional workload.

Table 2 contains CICS monitoring data and shows that for various transaction types, there is a small benefit gained in reduced response times and processor utilization using a `gencon` GC policy.

Table 2 CICS transaction times with different GC policies

Transaction type	**Mean response (ms)**		**Maximum response (ms)**		**Mean processor (ms)**	
	gencon	optthruput	gencon	optthruput	gencon	optthruput
HelloWorld	0.855	0.902	5.900	16.049	0.235	0.238
100 FC requests	2.078	2.124	14.154	16.825	0.979	0.997
Intensive maths	12.945	13.004	19.077	19.172	12.108	12.121
Intensive character processing	12.520	12.784	24.640	26.862	11.636	11.883

The graph in Figure 15 shows JVM garbage collection pause times as shown in the Garbage Collection and Memory Visualization tool. The graph illustrates data for the same workloads shown in Table 2 on page 19. Longer pause times result from using the `optthruput` policy. Even with the `gencon` policy, some infrequent long pauses occur, which are caused when the tenured area is filled and requires collection. Nursery collections are generally much shorter.

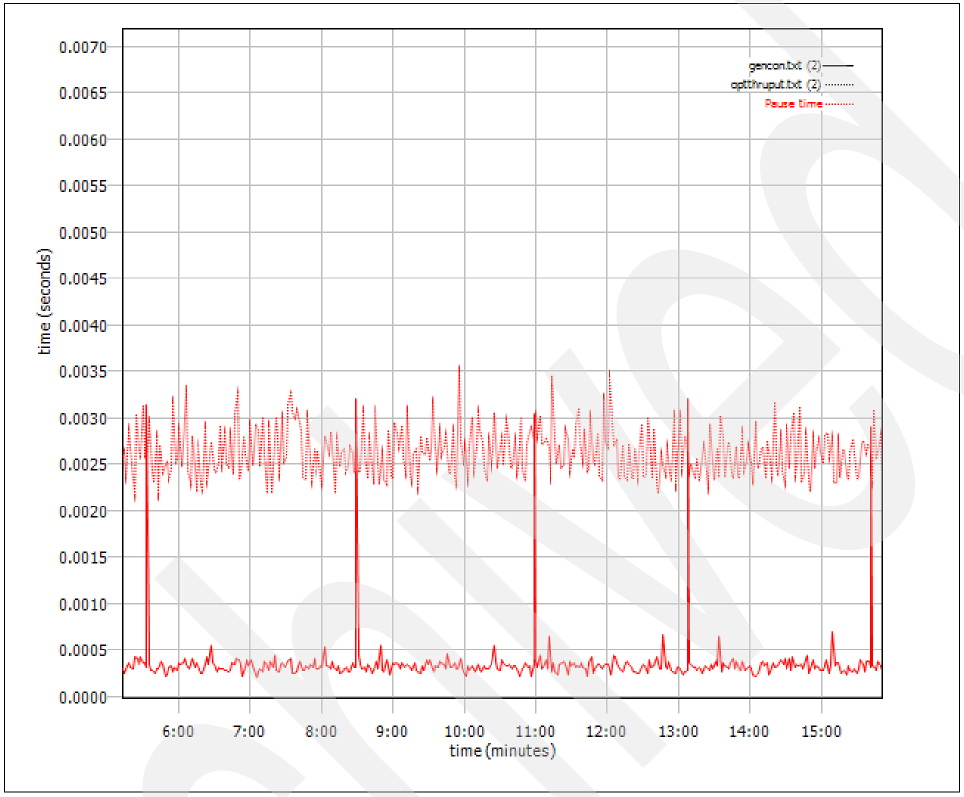


Figure 15 Comparing GC pause times using optthruput with gencon GC policies

Paying for a JVM run in CICS

The function that pays the processor resource cost depends on the type of action being performed.

- Startup

  The processor cost of starting a pooled JVM is paid by the user transaction that runs the first Java application in a JVM which triggers its creation. A CJSR transaction supplied by CICS is started when the JVM server resource is installed and pays the cost of starting a JVM server.

- Garbage collection

  A pooled JVM triggers garbage collection according to the percentage heap use setting **`GC_HEAP_THRESHOLD`** parameter, which can be added to the JVM profile. When the Java heap reaches this occupancy threshold, a CJGC transaction starts that forces a garbage collection within the pooled JVM. This mechanism is not suitable for the multi-threaded JVM server, which instead uses the GC triggering mechanism of the JVM. Normally, insufficient free space in the Java heap to allocate a new object, triggers GC. Also, an explicit call, such as using `System.gc()` in the Java application, can trigger GC.

In a JVM server, the CICS T8 TCB running the application that triggered the GC and a number of GC helper threads which run on TCBs other than CICS, share the processor cost of garbage collection. Only the processor resources used by the CICS T8 TCB is included in the processor usage value of the triggering CICS transaction reported by CICS monitoring. Total processor usage of the CICS address space includes the processor usage of the GC helper threads.

- Java just-in-time compilation

  For both the pooled JVM and the JVM server, the JIT runs on a TCB not supplied by CICS, so any processor resources used are not charged to a CICS transaction. The total processor usage of the CICS address space includes the processor cost.

Using 64-bit compressed references

Standard 64-bit Java object headers and references are twice the size of their 31-bit equivalents, as shown in Figure 16. Larger 64-bit objects more rapidly fill Java heap space and trigger garbage collection more frequently than their 31-bit equivalents. Maintaining the same frequency of garbage collection when moving from a 31-bit to 64-bit JVM, normally requires enlargement of the heap. Using the -Xcompressedrefs option of the 64-bit JVM halves the size of object headers and references, making them the same size as for the 31-bit JVM. This mode is restricted to Java heap sizes up to 29 Gb.

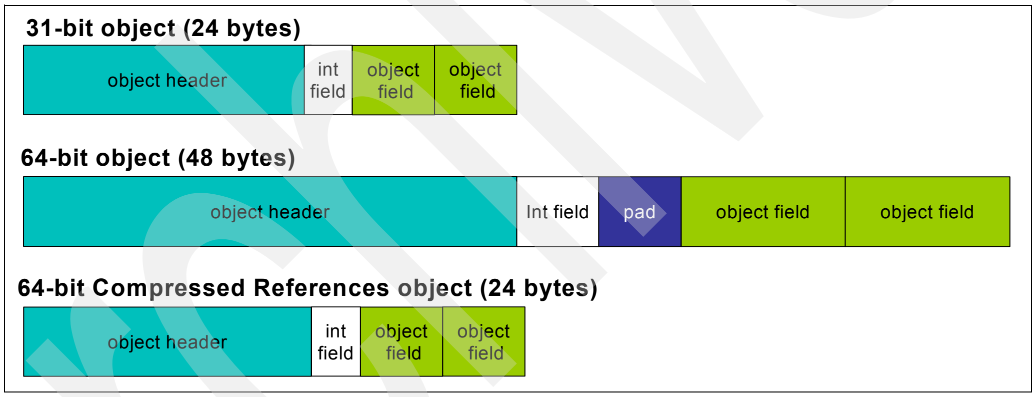


Figure 16 Example Java object size

The 64-bit JVM does need to place some structures in 31-bit memory when using compressed references, such as the modifiable class constant pool, JVM monitors, and Java threads. Thus, 31-bit memory used running in this mode slightly increases, compared to the normal 64-bit mode. For example, running the Java sample applications supplied by CICS, uses less than 1 MB more 31-bit memory running in compressed reference mode, in contrast to normal mode. To minimize the amount of 31-bit memory used by using compressed references, you must use Java V6.0.1 SR1 or later and add these parameters to the JVM profile used by the JVM server:

- `-Xcompressedrefs`
- `-XXnosuballoc32bitmem`

The benefits of using compressed references depend on the number of object references that exist in your application. As an example, Table 3 shows the difference in GC behavior for a simple CICS application that performs 100 file reads. For all three cases in Table 3, the same workload injection rate was used (approximately 470 transactions per second). In the first two sets of measurements, the heap was fixed at 30 MB with a fixed 25 MB nursery and fixed 5 MB tenured area. The third set of results shows the same workload using compressed references but with a smaller heap. In this case, the frequency of garbage collections is the same for the non-compressed references with a larger heap and improves the pause times and transaction processor costs.

Table 3 Comparing GC activity with and without compressed references

	Non-CR 30M heap	**CR 30M heap**	**CR 27M heap**
Number of garbage collections in 1 minute	112	100	116
Mean GC pause time (ms)	0.38	0.43	0.31
Amount of data flipped (MB per minute)	17.3	18.0	18.0
% time spent in GC pauses	0.08	0.08	0.06
Rate of garbage collection (MB per minute)	2475	2203	2216
Mean processor cost per transaction (ms)	1.003	0.998	0.986

The GC data for the Table 3 was extracted from the log data generated using the JVM **`-Xverbosegclog:`** parameter and analyzed using the Garbage Collection and Memory Visualizer tool provided as part of the IBM Support Assistant. The mean processor cost is the average USRCPUT data provided in CICS performance monitoring records.

Tuning the IBM Language Environment

Language Environment manages the UNIX program and memory environment for z/OS. A JVM manages the Java program and memory environment for z/OS. The JVM for z/OS is itself a C program. When a JVM is started, it runs as a C program within a Language Environment enclave. Similar to the way Language Environment supports UNIX programs, the JVM supports Java programs by providing program stacks and heap for any memory allocations that the Java programs make.

The Language Environment stacks and heaps are not in the same areas as the Java stacks and heap. The JVM creates the Java stacks and heap, along with any other memory allocations made by the JVM, such as class files or JIT buffers, and are allocated within an Language Environment heap. Figure 17 shows how some of these JVM areas are allocated within the memory areas of a Language Environment enclave.

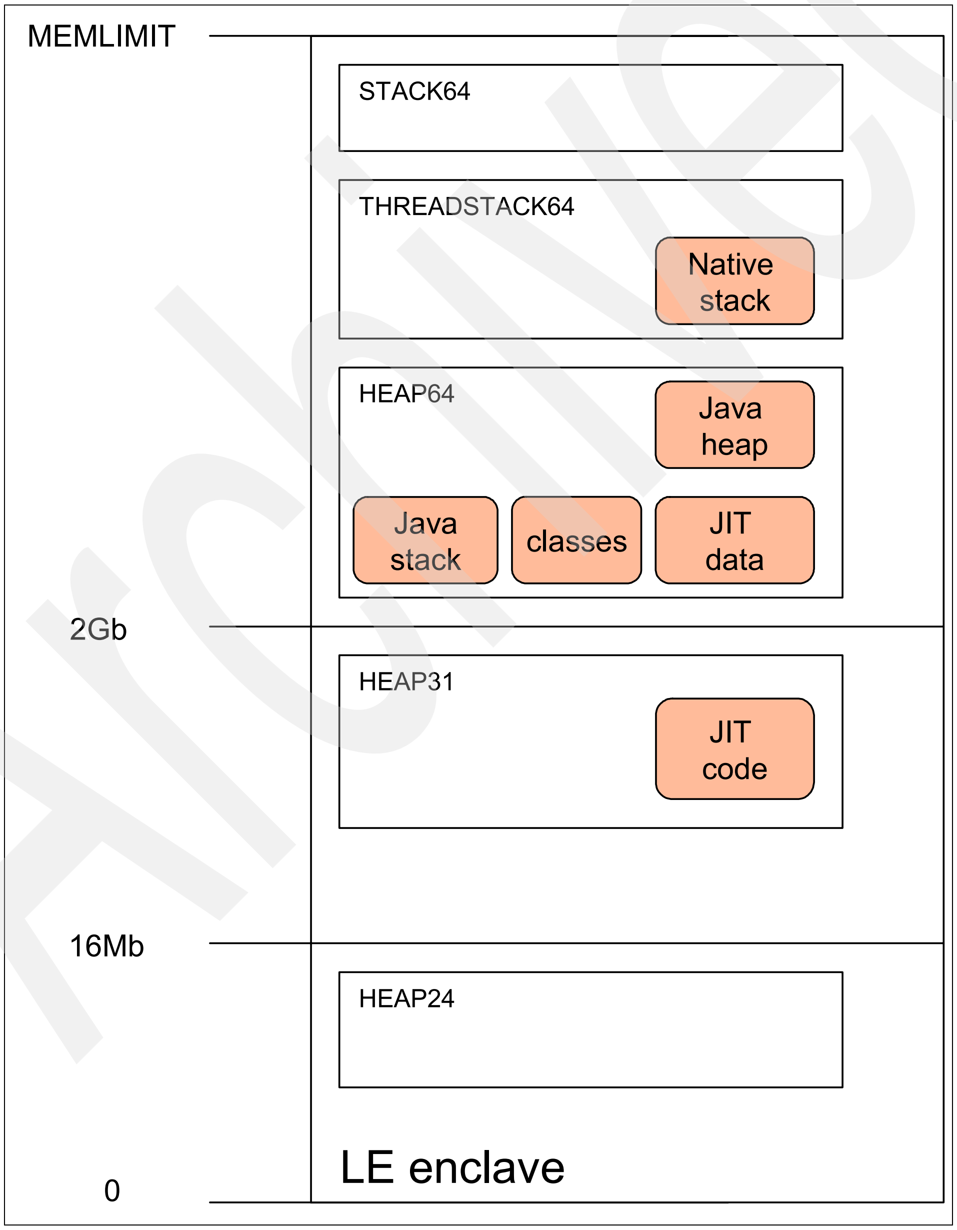


Figure 17 64-bit JVM areas within a Language Environment enclave

The specification supplied by CICS of the Language Environment stacks and heaps suit most cases, but if you are planning on using Java heaps in excess of 50 MB, then you probably benefit from changing the default settings.

The Language Environment options for both pooled JVMs and JVM servers are set by parameters compiled into load modules. For pooled JVMs, this load module is the DFHJVMRO user-replaceable module. For JVM servers, you can choose the module name which is specified in the JVMSERVER attribute **Lerunopts**. The default for **Lerunopts** is DFHAXRO. The SDFHSAMP data set provides the source of both DFHJVMRO and DFHAXRO.

To obtain a Language Environment statistics report, you must compile the Language Environment options module, including the source statement **RPTS** set to ON. This compilation causes a report to be written to stderr when the Language Environment enclave is closed, which occurs when the JVM is terminated. We do not suggest collecting statistics for normal operational use with **RPTS** set to ON as it requires significant, additional processor resources.

The Language Environment options load module DFHAXRO supplied by CICS for JVM server sets an initial 64-bit heap of 100 MB and initial 31-bit heap of 4 MB. Both areas allow incremental expansion to increase these initial allocations. Both areas can dynamically resize, but you gain best performance by setting KEEP for these increments. We suggest using the **HP64** and **HEAPP** Language Environment options, which are designed to improve the performance of multi-threaded applications

Figure 18 is an extract from the source of the DFHAXRO supplied by CICS and shows how the parameters are coded.

```
* Heap storage
         DC    C'HEAP64(100M,'          Initial 64bit heap
         DC    C'4M,'                   64bit Heap increment
         DC    C'KEEP,'                 64bit Increments kept
         DC    C'4M,'                   Initial 31bit heap
         DC    C'512K,'                 31bit Heap increment
         DC    C'KEEP,'                 31bit Increments kept
         DC    C'1K,'                   Initial 24bit heap
         DC    C'1K,'                   24bit Heap increment
         DC    C'KEEP) '                24bit Increments kept
* Heap pools
         DC    C'HP64(ALIGN) '
         DC    C'HEAPP(ALIGN) '
```

Figure 18 Extract from Language Environment options supplied by CICS

The statistics report output is explained in some detail in the publication *z/OS: Language Environment Debugging Guide*, GA22-7560-08, available at the following site:

http://publib.boulder.ibm.com/infocenter/zvm/v5r4/index.jsp?topic=/com.ibm.zvm.v54.hcpa0/hcsf8b32140.htm

Taking advantage of zAAP and zIIP

Special purpose processors are available on all IBM zEnterprise™, System z10®, and System z9® servers. Among these special purpose processors are System z Application Assist Processors (zAAP) and System z Integrated Information Processors (zIIP). These speciality processors are designed to reduce the total cost of computing.

For CICS workloads, the processors provide alternatives to the standard processors to run Java and z/OS XML System Services requests. The zAAP on zIIP capability further optimizes performance and make it possible for workloads eligible for zAAP to run on a zIIP so that only one type of speciality engine is needed.

zAAP on zIIP functions are described in detail at the following site:

`http://www.ibm.com/systems/z/hardware/features/ziip/index.html`

Not all of the work run by a JVM can run on a zAAP or zIIP. For example, execution of native methods using the Java Native Interface (JNI) is not eligible. This mechanism is commonly used within CICS to provide access to CICS services from a Java application. Although specific TCBs are used to run Java code within CICS (T8 for JVM server and J8 or J9 for JVMPOOL), zAAPs or zIIP cannot provide all the processor time these TCBs require.

RMF can be used to report both the expected and actual usage of zAAP and zIIP processors. Setting **PROJECTCPU=YES** in the IEAOPTxx parameters also makes it possible for RMF to report on zAAP and zIIP-eligible work that runs on a standard processor.

The following data was collected on a zEnterprise system with three standard processors and one zIIP. In the first RMF WLMGL workload activity report (Figure 19), the three standard processors are online and the zIIP is offline. The CICS workload is running at a transaction rate of 117.20 transactions completed per second (reported in the *END/S* field in the RMF report) and uses 140.79% of standard processor time. (300% is available on this zEnterprise system, as reported in the *CP* field in APPL% section). Of this 140.79%, RMF reports that 134.47% is eligible for running on a zIIP (shown in the *IIPCP* field of the RMF report). This value implies that 6.32% of the processor time used is not eligible to run on a zIIP, so it must run on a standard processor.

See *RMF Report Analysis* manual (SC33-7991-17) for full details.

```
REPORT BY: POLICY=POLICY                         REPORT CLASS=CICS2A20
                                                 DESCRIPTION =

-TRANSACTIONS-  TRANS-TIME HHH.MM.SS.TTT  --DASD I/O--  ---SERVICE---   SERVICE TIME   ---APPL %---
AVG       1.00  ACTUAL                15  SSCHRT   0.0  IOC         10  CPU   84.156   CP    140.79
MPL       1.00  EXECUTION              2  RESP    52.0  CPU     47918K  SRB    0.317   AAPCP   0.00
ENDED     7032  QUEUED                 0  CONN    32.5  MSO      5644M  RCT    0.000   IIPCP 134.47
END/S   117.20  R/S AFFIN              0  DISC    19.5  SRB     180328  IIT    0.000
#SWAPS       0  INELIGIBLE             0  Q+PEND   0.0  TOT      5692M  HST    0.000   AAP      N/A
EXCTD        0  CONVERSION             0  IOSQ     0.0  /SEC    94876K  AAP      N/A   IIP     0.00
AVG ENC   0.00  STD DEV              212                                IIP    0.000
REM ENC   0.00                                          ABSRPTN    95M
MS ENC    0.00                                          TRX SERV   95M
```

Figure 19   RMF report with only three standard processors

The data shown in Figure 20 was collected after one zIIP was varied online. From the report shown in Figure 19 on page 25, 134.47% of standard processor time is eligible for offloading to a zIIP. However, this report shows that only 91.48% is run on a zIIP (shown in `IIP` field in the APPL% section). The standard processors are still 53.07% busy, and 45.08% of that time is still eligible for offload. This situation occurs with concurrent workloads when multiple zIIP-eligible tasks are running, but not enough zIIP processors are available to run them all simultaneously.

```
REPORT BY: POLICY=POLICY                         REPORT CLASS=CICS2A20
                                                 DESCRIPTION =

-TRANSACTIONS-  TRANS-TIME HHH.MM.SS.TTT  --DASD I/O--  ---SERVICE---   SERVICE TIME   ---APPL %---
AVG       1.00  ACTUAL                60  SSCHRT   0.0  IOC         0   CPU   86.274  CP     53.07
MPL       1.00  EXECUTION              7  RESP     0.0  CPU    46952K   SRB    0.455  AAPCP   0.00
ENDED     7036  QUEUED                 0  CONN     0.0  MSO     7707M   RCT    0.000  IIPCP  45.08
END/S   117.27  R/S AFFIN              0  DISC     0.0  SRB    247359   IIT    0.000
#SWAPS       2  INELIGIBLE             0  Q+PEND   0.0  TOT     7754M   HST    0.000  AAP      N/A
EXCTD        0  CONVERSION             0  IOSQ     0.0  /SEC  129239K   AAP      N/A  IIP    91.48
AVG ENC   0.00  STD DEV              573                                IIP   54.886
REM ENC   0.00                                          ABSRPTN   129M
MS ENC    0.00                                          TRX SERV  129M
```

Figure 20 RMF report with only three standard processors and one zIIP

Using the IEAOPTxx parameter **`IIPHONORPRIORITY=NO`** forces all zIIP-eligible work to run only on zIIP processors. The data in Figure 21 was collected with this setting with three standard processors and one zIIP online. The transaction rate is reduced from 117.27 to 85.85 transactions completed per second because all zIIP-eligible work must be run on a zIIP processor. The one zIIP processor available is fully used, which is reported in the *IIP* field of the report in Figure 21 as 99.47% busy. Increasing throughput requires either more zIIPs online.

```
REPORT BY: POLICY=POLICY                         REPORT CLASS=CICS2A20
                                                  DESCRIPTION =

-TRANSACTIONS-  TRANS-TIME HHH.MM.SS.TTT  --DASD I/O--  ---SERVICE---   SERVICE TIME   ---APPL %---
AVG       1.00  ACTUAL             1.461  SSCHRT   0.0  IOC         0   CPU   63.488  CP      6.92
MPL       1.00  EXECUTION              0  RESP     0.0  CPU    34552K   SRB    0.342  AAPCP   0.00
ENDED     5151  QUEUED                 0  CONN     0.0  MSO   617083K   RCT    0.000  IIPCP   0.00
END/S    85.85  R/S AFFIN              0  DISC     0.0  SRB    185975   IIT    0.000
#SWAPS       0  INELIGIBLE             0  Q+PEND   0.0  TOT   651821K   HST    0.000  AAP      N/A
EXCTD        0  CONVERSION             0  IOSQ     0.0  /SEC   10864K   AAP      N/A  IIP    99.47
AVG ENC   0.00  STD DEV            2.471                                IIP   59.680
REM ENC   0.00                                          ABSRPTN    11M
MS ENC    0.00                                          TRX SERV   11M
```

Figure 21 RMF report with IIPHONORPRIORITY=NO

Tools for investigating Java performance in CICS

In this section, we review the tools available to investigate Java performance in CICS:

- CICS statistics
- CICS monitoring
- Resource Measurement Facility (RMF)

CICS statistics

In addition to the pooled JVM reports available in previous releases, CICS TS V4.2 provides a JVM server statistics report that includes tuning information. Figure 22 shows that the JVM server JVMO1 is used 28,125 times in the measurement interval (in this case, set to 1 minute). The JVM server has a thread limit of 5, so it can handle a maximum of five concurrent requests to run Java programs. In this case, many more requests run concurrently than can be supported, as indicated by the `thread limit waits` and `peak thread limits wait` counts. Some transactions are waiting for a thread to become available. The thread limit can be increased to reduce this waiting period, improving response time and throughput, but doing so also increases demand for processor time.

In addition to the thread data, JVM server statistics provide a view of Java heap use. In this case, the Java heap was fixed to 30 MB by setting initial and maximum heap sizes to the same value. During the 1-minute measurement interval, the statistics report no major GC collections and 111 minor GC collections. The JVM server uses the generational **`-Xgcpolicy:`**`gencon` configuration, meaning that during this interval, only the nursery area needed to be collected, which happened 111 times. During the 1-minute interval, the JVM server spent a total of 41 ms in garbage collection. This figure is about 0.07% of the total elapsed time, which is a short time for a GC pause.

```
JVM server STATISTICS
____________________
JVM server Name . . . . . . . . . . :  JVMO1
JVM server JVM profile name . . . . :  jvmpropo
JVM server LE runtime options . . . :  DFHAXRO
JVM server use count. . . . . . . . :     28125
JVM server thread limit . . . . . . :         5
JVM server current threads. . . . . :         5
JVM server peak threads . . . . . . :         5
JVM server thread limit waits . . . :     61751
JVM server thread limit wait time . :  00:01:10.9228
JVM server current thread waits . . :        14
JVM server peak thread limit waits. :        14
JVM server system thread use count. :         1
JVM server system thread waits. . . :         0
JVM server system thread wait time. :  00:00:00.0000
JVM server current sys-thread waits.:         0
JVM server peak system thread waits :         0
JVM server state. . . . . . . . . . :  Enabled
JVM server JVM creation time. . . . :  10/19/2011 09:13:08.9865
JVM server current heap size . . . .:  26147616
JVM server initial heap size . . . .:       30M
JVM server maximum heap size . . . .:       30M
JVM server peak heap size . . . . . :  31334008
JVM server heap occupancy . . . . . :   4993200
JVM server Garbage Collection policy:  -Xgcpolicy:gencon
JVM server major GC collections. . .:         0
JVM server elapsed time in major GC.:         0
JVM server major GC heap freed . . .:         0
JVM server minor GC collections. . .:       111
JVM server elapsed time in minor GC.:        41
JVM server minor GC heap freed . . .:     2743M
```

Figure 22   Example CICS statistics for JVM server

CICS monitoring

Monitoring data provides detailed information for a CICS task and can be readily analyzed using the utility DFH$MOLS supplied by CICS or CICS Performance Analyzer for z/OS. More information about this tool is available from this site:

http://www.ibm.com/software/htp/cics/panaly/

Some monitor fields are useful for reviewing the performance of CICS Java applications using a JVM server, as follows:

- **T8CPUT**

  Processor time used running in a JVM server.

- **JVMTHDWT**

  Elapsed time the user task waited to obtain a JVM server thread because the CICS system reached the thread limit for a JVM server in the CICS region.

- **MAXTTDLY**

  Elapsed time waiting to obtain a T8 TCB because CICS reached the limit of available TCBs. A limit of 1024 T8 TCBs exists for each CICS region, and each JVM server within a CICS region can use up to 256 T8 TCBs concurrently.

Resource Measurement Facility

RMF can report processor use and transaction rate data for a single or groups of CICS regions. In Figure 23, an RMF reporting class was defined using the z/OS Workload Manager (WLM) administrative application for a region called CICS2A20. WLM classification rules were added for this region name to both CICS and JES subsystems so that transaction and processor data is shown in the CICS2A20 report class.

The RMF report in Figure 23 shows that 28,051 transactions ended during the 59.853-second reporting interval, producing a transaction rate of 468.66 transactions per second. The processor resources used by the entire CICS region were 51.64% of standard processor time and 5.41% of special purpose zIIP processor time. This report gives a useful view of total resource consumption of a CICS region, including all the processor usage costs that are included in CICS monitoring data.

```
REPORT BY: POLICY=POLICY                          REPORT CLASS=CICS2A20
                                                  DESCRIPTION =

-TRANSACTIONS-  TRANS-TIME HHH.MM.SS.TTT  --DASD I/O--  ---SERVICE---   SERVICE TIME   ---APPL %---
AVG       1.00  ACTUAL                 7  SSCHRT   0.0  IOC       7395  CPU    33.176  CP      51.64
MPL       1.00  EXECUTION              1  RESP     0.0  CPU     18890K  SRB     0.966  AAPCP    0.00
ENDED    28051  QUEUED                 0  CONN     0.0  MSO      6753M  RCT     0.000  IIPCP    0.22
END/S   468.66  R/S AFFIN              0  DISC     0.0  SRB     549823  IIT     0.000
#SWAPS       2  INELIGIBLE             0  Q+PEND   0.0  TOT      6773M  HST     0.000  AAP       N/A
EXCTD        0  CONVERSION             0  IOSQ     0.0  /SEC   113154K  AAP       N/A  IIP      5.41
AVG ENC   0.00  STD DEV              307                                IIP     3.237
REM ENC   0.00                                          ABSRPTN   113M
MS ENC    0.00                                          TRX SERV  113M
```

Figure 23 Example RMF WLMGL report

Java tools from IBM Support Assistant

The IBM Support Assistant is a complementary software offering that includes various serviceability tools for many IBM software products, including Java. For details about the Support Assistant, see the following site:

`http://www.ibm.com/software/support/isa/`

Along with the IBM Monitoring and Diagnostics Tools for Java provided by the IBM Support Assistant, is the Health Center tool. This tool is composed of two parts:

- A low-cost agent running within a monitored JVM
- An Eclipse-based client that connects to the agent to collect, analyze, and display monitor data from that JVM

The displays are presented as a number of real-time views covering different aspects of the JVM. Among these views, Garbage collection is useful for tuning the JVM heap, and Profiling provides a sampling profile of Java methods, including call paths, to help analyze the performance of Java applications.

You can also use the Garbage Collection and Memory Visualizer tool to tune the JVM heap. This tool uses JVM log data to provide a graphical view of a wide range of garbage collection data. An example is shown in “Choosing JVM garbage collection policies” on page 19.

Further reading

Further information about CICS and Java can be found at the following sites:

- CICS Information Center

  `https://publib.boulder.ibm.com/infocenter/cicsts/v4r2/index.jsp`
- Enhance performance with class sharing

  `http://www.ibm.com/developerworks/java/library/j-sharedclasses/`
- IBM Support Assistant

  `http://www.ibm.com/software/support/isa/`
- Match 31-bit IBM WebSphere® Application Server performance with new features in 64-bit Java on System z

  `https://www.ibm.com/partnerworld/wps/servlet/ContentHandler/whitepaper/systemz/java_websphere/performance`
- *z/OS 64-bit IBM MVS Programming: Extended Addressability Guide,* SA22-7614-07

  `http://publibz.boulder.ibm.com/epubs/pdf/iea2a570.pdf`

The person who wrote this paper

This paper was produced by a specialist working at the International Technical Support Organization, Hursley Center.

Graham Rawson is a CICS Performance Analyst within the CICS Development group based at the IBM Laboratory in Hursley, England. Graham has worked for IBM for 26 years in various roles supporting CICS Transaction Server and IBM Java technology. Graham holds a B.Sc. (Hons) in Chemistry from the University of East Anglia (Norwich, England) and a Certificate in Software Engineering from the University of Oxford (England).

Notices

This document REDP-4850-00 was created or updated on March 29, 2012.

Send us your comments in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
- Send your comments in an email to:
  redbooks@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Redpaper™

Trademarks

CICS®	Redbooks (logo) ®	WebSphere®
IBM®	Resource Measurement Facility™	z/OS®
Language Environment®	RMF™	z10™
MVS™	System z10®	z9®
Redbooks®	System z9®	zEnterprise™
Redpaper™	System z®	zSeries®