

## **Basic Java Performance for iSeries (V5R1 Group SF99069-13 and Beyond)**



William Berg, Arvin Fisher, Dan Hicks  
iSeries Rochester Lab

This white paper is intended to be an “executive summary” level of information relative to Java™ on the iSeries server after the introduction of MMI (Mixed Mode Interpreter) with Group PTF SF99069-13 in December of 2002. The basic message is that with the advent of that PTF, and actions on V5R2 and future releases, getting the best performance out of Java becomes more “autonomic” than it ever has been in the past.

## Introduction

There are a lot of areas that are worth tuning to get the best Java performance possible, but this paper is about “the big hitters” the items that have a good chance at giving your application a performance boost that is noticeable. Not surprisingly, the areas that will be touched on are code generation, memory usage, and startup considerations.

Code generation has been a focus area for Java since the language first hit the scene as an interpreted language. Just In Time (JIT) compilers were quick to arrive, and on the iSeries, Static Compilation, what we like to call “DE” for Direct Execution. With the latest PTF levels on the iSeries, the JIT Mixed Mode Interpreter (MMI) is now the fastest execution mode available. It is also the default, so for programs that have never been run through CRTJVAPGM, this improvement may already be realized without any configuration changes. We will explain MMI and its usage in more detail.

Processors are faster than ever, so the relative cost of waiting for a page fault is also at new all-time highs. Java is an object oriented garbage collected (GC) language, so it tends to not only be memory hungry, but have poor locality of reference when GC runs. We will discuss the positive aspects of running Java in its own pool to prevent excessive paging during GC cycles which can cause very poor performance.

As Java has advanced, more and more dynamic approaches have been invented, including Java Server Pages where it is difficult for the JVM to maintain any relationship between a “file” (or other persistent object), and the Java class. Since the iSeries server needs to verify all classes, this can mean that the overhead of verification is significantly increased from a “one time cost” to a “once per JVM startup” cost. We will discuss the verification cache, activated through use of the **os400.define.class.cache.file** property to alleviate this problem.

## Mixed Mode Interpreter (MMI)

Prior to Group SF99069-13 on V5R1 the JIT was generally slower than programs that had been optimized through the Java Transformer at optimization level 40. JVM folks tend to refer to transformed programs as “Direct Execution”, or DE, so we will use that from here on out. Even as the runtime performance of Java Programs optimized with the JIT edged closer to DE performance, the JIT programs paid a large penalty in startup performance over DE, on the order of 4-6x. For smaller programs this was often not noticeable, but for a large application like Websphere, the startup time would be tens of minutes longer, and completely unacceptable.

With the advent of MMI, the startup cost over DE has been reduced to 15-20%, and the runtime performance of the JIT is now 15-20% faster than DE Op 40 in many applications. While the technology to do this is quite sophisticated, the principle is very simple. MMI works as follows:

- Java methods are interpreted for the first “N” executions.
- During those N executions, profile information about how the program is running is gathered.
- When the counter hits N, the method is run through the JIT, and the JIT uses the profile information to do a better job of optimization.

Some will have a desire to “optimize N”, and indeed upon first blush, the default value of 2K that has been picked for N sounds too big to almost everyone. While there may be some cases where performance can be gained by changing that number by changing **os400.jit.mmi.threshold**, in general that impulse should be resisted for a couple of reasons:

1. As the technology evolves, we may change the default to better consider new realities. Programs running with a property specified will not gain the advantage of that new information.
2. This value is a great value to become “autonomic”. By gathering and storing data relative to usage of methods, we may well be able to dynamically “learn” the optimum value. Again, we will be forced to honor specific threshold values specified.

If however, you simply must, smaller values will usually result in both slower startup and slower runtime performance (less information gathered, less performance in the generated code), but at the potential advantage of more predictable transaction times once the JIT has done its work. Higher values will tend to improve runtime performance incrementally (once the JIT has done its work), and have no noticeable effect on startup. However, setting the value too high means that a lot of code is being interpreted until the threshold is reached.

## Pool Size

Java is an object oriented language, so it creates objects in storage. Generally a lot of objects. What is more, Java provides automatic garbage collection (GC), which means that some time after an object is no longer able to be accessed by any running Java code, the Garbage Collector will make sure it is deleted. The iSeries server allows the amount of real (as opposed to virtual) memory to be specified by a Memory Pool. Page faults occur whenever the actual size of your GC heap, and any other programs using memory from that pool if the pool is shared, exceed the amount of real memory specified in the pool.

Excessive page faults may occur if the memory pool for your JVM is too small. These faults will be reported as non-database page faults on the WRKSYSSTS command display. Typically, the storage pool for your JVM is \*BASE. Fault rates between 20 and 30 per second are generally acceptable, but higher rates should be reduced by increasing

memory pool size. In some cases, reducing this value below 20 or 30 per second may improve performance as well. Lowering the GCHINL parameter might also reduce paging rates by reducing the OS/400 JVM heap size, but may also cause some performance problems due to more frequent GC cycles.

A memory pool also has an activity level associated with it which specifies the number of threads that can actively use processor(s) at the same time from that memory pool. When more threads are started than are allowed to concurrently execute due to the activity level control, the excess threads will be forced to wait for an available activity level slot before they can run. The number of threads running (active threads) refers to the number of threads that are eligible to compete for a processor and that count against the activity level for a memory pool. Active threads do not include threads that are waiting for input, for a message, for a device to be allocated, or for a file to be opened. Active threads do not include threads that are ineligible (threads that are ready to run but the memory pool activity level is at its maximum).

Once the maximum activity level for a memory pool has been reached, additional threads needing the memory pool are placed in the ineligible state. The threads wait in ineligible state for the number of active threads in the memory pool to fall below the maximum activity level, or for a thread to reach the end of its time slice. As soon as a thread gives up its use of the memory pool, the other threads that are not active become eligible, and will be dispatched based on their priority. Having Java threads in the ineligible state can cause severe performance degradation as well as excessive JVM GC heap growth. When the Java threads are in the ineligible state they are unable to communicate with the Garbage Collector, which will prevent a Java GC cycle from completing. This will cause the GC heap to grow rapidly, which will tend to drive the pool into page thrashing, and further degrade performance.

In order to avoid this condition, set the activity level of the memory pool to at least as large as the maximum number of threads you expect to be concurrently active at any time for the JVM running in the memory pool.

## Verification Cache

One downside of being the only JVM that is integrated into the operating system is that there is really no choice but to verify the Java programs that we run. In a server environment, we would argue that everyone would likely want to do verification if they realized that it was often being bypassed on other machines, but this is yet another case of most people feeling very good about what they don't know, and very bad about a JVM that has the audacity to force something that is being ignored elsewhere.

One of the great things about DE was that the price of creating the program was so high, that the cost of verification was completely invisible. When code like Java Server Pages (JSPs) showed up however, even after solidly improving the performance of our verifier, it became quite clear that "zero" (the cost incurred by those skipping verification) was going

to be a really hard number to beat with even the worlds fastest verifier. In order to reduce our cost of verification, the verification cache was developed.

Before going much deeper here, this problem is only a problem if the startup time of your application is your concern, AND you know (or have reason to believe) that you have classes that are being loaded dynamically by a user class loader, AND if it's likely that the JVM is unable to maintain a linkage to the JVAPGM for those classes (because something other than the standard URLClassLoader mechanism is being used).

The "verification cache" operates by caching JVAPGMs that have been dynamically created for dynamically loaded classes. When the verification cache is not operating, these JVAPGMs are created as temporary objects, and are deleted as the JVM shuts down. When the verification cache is enabled, however, these JVAPGMs are created as persistent objects, and are cached in the (user specified) machine-wide cache. If the same (byte-for-byte identical) class is dynamically loaded a second time (even after the machine is re-IPLed), the cached JVAPGM for that class is located in the cache and reused, eliminating the need to verify the class and create a new JVAPGM (and eliminating the time and performance impact that would be required for these actions). Older JVAPMGs are "aged out" of the cache if they are not used within a given period of time (default is one week).

In general, the only cost of enabling the verification cache is a modest amount of disk space. If it turns out that your application is not using one of the problem user class loaders, the cache will have no impact, positive or negative, while if your application is using such a class loader then the time taken to create and cache the persistent JVAPGM is only slightly more than the time required to create a temporary JVAPGM. With next to zero downside risk, and a decent potential to improve performance, the verification cache is well worth a try, and will likely become the default in the near future.

Maintenance is not a problem either: if the source for a cached JVAPGM is changed, the currently-cached version will simply "age out" (since its class will no longer be a byte-for-byte match), and a new JVAPGM will be silently created and cached. Likewise, the cache doesn't care about JDK versions, PTFs installed, application upgrades, etc. Aside from specifying a valid value (eg, /QIBM/ProdData/Java400/QDefineClassCache.jar) for os400.define.class.cache.file to enable the verification cache in the first place, the only other thing you may need to do is set **os400.define.class.cache.maxpgms** to a value of, say 20000, since the default of 5000 had been shown to be too small for many applications.

## Conclusion

The iSeries JVM team will continue to improve on the Java technology so that iSeries server remains a premier platform for running Java with as little work required on the part of the customer as possible. To that end, it is likely that the verification cache will become the default in a future release and we will continue to improve GC so that it is easier to see when problems arise through tooling like iDoctor, and where possible, for us to adjust the GC algorithms to optimize performance without customer impact. In the meantime, the technologies discussed here are all available by installing the latest group PTF on V5R1, or by moving to V5R2 where they are integrated in the base. We hope this discussion will be helpful to you in improving the performance of your iSeries Java applications.

## Author Contacts and Additional Information

For questions regarding the content and conclusions drawn in this report, please contact Bill Berg, [bilber@us.ibm.com](mailto:bilber@us.ibm.com), or Richard Odell, [rjodell@us.ibm.com](mailto:rjodell@us.ibm.com).

Additional resources can be found in the following websites:

- **Redbooks** at: <http://www.redbooks.ibm.com/>
  - **Java and WebSphere Performance on IBM iSeries Servers**  
[http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246256.html?](http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246256.html?Open)  
[Open](#)
  - **Java and Websphere Performance on iSeries Servers, SG24-6256-00**  
[http://publib-b.boulder.ibm.com/Redbooks.nsf/RedpieceAbstracts/sg246256.html?](http://publib-b.boulder.ibm.com/Redbooks.nsf/RedpieceAbstracts/sg246256.html?Open)  
[Open](#)
- **Just-in-Time Execution in Java** article  
<http://www.as400network.com/article.cfm?ID=7759>
- **V5R2 Performance Capabilities Reference Manual**. See Chapter 7 for Java Performance found at:  
<http://www-1.ibm.com/servers/eserver/series/perfmgmt/resource.htm>
- **Workload Estimator** (Java Workload) web site:  
<http://www-1.ibm.com/eserver/series/support/estimator>
- **WebSphere Performance Considerations** at:  
[http://www-919.ibm.com/developer/performance/pdf/was5\\_performance.pdf](http://www-919.ibm.com/developer/performance/pdf/was5_performance.pdf)

## **Disclaimer - System Performance**

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Information is provided "AS IS" without warranty of any kind. Mention or reference to non-IBM products is for informational purposes only and does not constitute an endorsement of such products by IBM.

The material included in this presentation regarding third parties is based on information obtained from such parties. No effort has been made to independently verify the accuracy of the information. This presentation does not constitute an expressed or implied recommendation or endorsement by IBM of any third party product or service.

The information in this presentation is based on publicly available information about Microsoft and the Windows 2000 products, Compaq products, and Sun products. The information contained in this presentation is believed to be accurate by the author on the date it was published. However IBM does not warrant or guarantee the accuracy of the information.

The information presented regarding unannounced IBM products is a good faith effort to discuss iSeries plans and directions. IBM does not guarantee that these products will be announced.

## Trademarks

© Copyright International Business Machines Corporation 2003. All rights reserved. References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both:

ADSTAR	Client Series	JustMail	PSF
Advanced Function Printing	DataGuide	Net.Commerce	SanFrancisco
AFP	DB2 Universal Database	Net.Data	SmoothStart
AIX	e-business logo	Netfinity	SystemView
AnyNet	IBM	NetView	WebSphere
Application Development	IBM Logo	OfficeVision	
APPN	IBM Network Station	OS/2	
AS/400	Information Warehouse	Operating System/400	
AS/400e	Integrated Language Environment	OS/400	
AT	Intelligent Printer Data Stream	PowerPC	
BrioQuery	IPDS	PowerPC AS	
BRMS	iSeries	Print Service Facility	

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information in this presentation concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources. Sources for non-IBM list prices and performance numbers are taken from publicly available information. Including vendor announcements, vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdraw without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.