# IBM Rewrites the Book on Java™ Performance

## Introduction

The Java programming environment has long been recognized for improving programmer productivity through its "Write Once, Run Anywhere" architecture independence, object oriented data model, and reusable bean components. It is also touted for its enablement of applications which are, at the same time, multi-threaded, interactive, safe, and robust. ***Now, Java programs can be noted for their speed of execution as well.*** In May 2000, IBM has announced a new family of Java™ 2 version 1.3 deployment implementations for six operating system platforms: Windows®, Linux, AIX, OS/2, OS/400, and S/390. Previews of the Linux and AIX versions are available immediately for download from IBM's alphaWorks (*http://www.alphaworks.ibm.com*).

IBM Just-In-Time (JIT) compiler technology, which comes directly from its world renowned Research Laboratory in Tokyo, Japan, uses state-of-the-art compilation techniques to provide performance often as good as or better than traditional programming languages. Common object-oriented programming practices require a Java Virtual Machine (JVM) to manage the allocation of objects, the collection of garbage (i.e. objects which are no longer needed), and the synchronization of method calls. These features often become the source of performance bottlenecks in Java applications. IBM has carefully engineered its implementation of JVM to eliminate these bottlenecks, and enable more CPU time to be spent on the real work of the Java application. In addition, improvements in the implementation of the Java Native Interface and networking libraries increase performance of Java application performance when integrated with other computer subsystems. For mission-critical applications, predictability of performance is often a crucial characteristic. For some JVM implementations observed in the industry, this has been problematic. SPECjvm98 scores, for instance, can vary by a factor of 3 from one run to another in some of the highest postings found at *http://www.spec.org/osg/jvm98.* IBM has improved predictability of performance by introducing its Mixed Mode Interpreter (MMI). MMI technology determines which methods to compile and when to compile them, thereby eliminating unnecessary overhead in compiling methods that will only be executed a few times.

This paper will discuss unique features of the IBM implementation of Java 2 technology that are designed to deliver outstanding Java application performance. To illustrate the capacity for performance improvement, results from three industry standard benchmarks will be provided on four of the recently announced operating system platforms: Windows, Linux, AIX, and OS/400. For example,

- The SPECjvm98 scores obtained from IBM implementation of Java 2

version 1.3 for Windows and Linux platforms represent the highest scores in the benchmark 48-256 MB category.

- The VolanoMark score for IBM implementation of Java 2 version 1.3 for Linux platform is 68% better compared to implementation of Java 1.1.8 Linux.
- On the VolanoMark network scalability test with 9,000 connections, an 8-way RS/6000 M80 with 4GB of memory running Java 2 version 1.2.2 and AIX 4.3.3, transferred 11,960 messages per second, 147 percent faster than the Sun E6500's 4,847 messages per second.
- The AS/400 Model 840-2420, powered by IBM’s I-Star (copper and silicon-on-insulator) processors and running IBM’s implementation of Java 2 version 1.3 on OS/400 V4R5, delivered a record VolanoMark score of 108,153 messages persecond.
- The AS/400 Model 840-2420 with IBM’s implementations of Java 2 version 1.2.2 and 1.3 on OS/400 V4R5 produced the highest SPECjbb2000 scores.

These results demonstrate that IBM’s implementations of Java 2 provide a solid Java foundation spanning a broad variety of platforms with remarkable performance improvement over previous releases.

**Java 2 Technology from IBM**

IBM’s implementations of Java 2 technology include the following unique features.

- JIT dynamic Java compiler with advanced Java optimization techniques
- Mixed Mode Interpreter for selective compilation
- Code base engineered for performance and stability
    - Fast object allocation
    - Streamlined garbage collection
    - Effective heap size management
    - Efficient synchronization (monitors)
    - Scalability
    - Fast Java Native Interface (JNI)
    - Efficient networking
    - Java 2 security

IBM’s implementations also support Java Virtual Machine Profiling Interface (JVMPI). As a result, Java performance and profiling tools vendors can use profiling hooks provided in our implementation to enable their JVM-independent tools.

While each feature discussed in the following sections is used by some or all of IBM's implementations of Java 2 technology, not all features are used on all platform implementations. In some cases, specific platform environments present

unique opportunities for "better" optimizations. In these cases, IBM has implemented different features that exploit the platform specific opportunities.

**The IBM Just-In-Time (JIT) Compiler**

IBM provides a compiler built into its Java runtime environment (JRE) which will take the platform-independent Java bytecode and translate it into optimized native machine instructions, saving the native code in memory for future execution during the life of the application. Using state-of-the-art compilation technology, the IBM Just-in-time compiler employs many traditional compiler optimizations specifically tuned to the Java environment. It also implements new optimizations specifically for object-oriented environments and several techniques to address specific features of the Java language as well as to exploit the run-time knowledge of the compiler. The result is a light-weight Java runtime compiler which provides an efficient execution environment with very low overhead.

As described above, the JIT compiler is only invoked for methods that are compute-intensive, so the time taken to perform these optimizations is more than recovered by the faster runtime execution. In addition, there are several Java-specific issues that the IBM JIT compilers optimizes :

- Frequent invocation of small virtual methods, due to standard object-oriented programming practices. The IBM JIT compiler employs aggressive inlining to optimize these methods, including inlining virtual methods where permitted (i.e., when the methods are not overridden).
- Runtime exception checking required by the Java language itself (e.g, array bounds checking). The IBM JIT compiler employs sophisticated optimizations to only perform these exception checks when required, thereby producing more optimal code.
- Runtime overhead of locking and unlocking to support synchronized methods. In addition to IBM's JVM optimizations producing a streamlined monitor implementation (described later in this paper), the IBM JIT compiler will also inline the most common synchronization paths in the JIT-compiled code, in order to avoid call/return overhead.

Major features for the IBM JIT compiler in its Java 2 V1.3 implementations are:

- Quadruple based internal representation and related optimizations. The quadruple representation is a set of information (source1, source2, destination operands, and opcode) annotated in each of the bytecode instructions. Data-flow analysis is done based on the quadruple.
- Common sub-expression elimination and deadcode elimination.
- Dag-based loop optimizations.
- Aggressive method devirtualization and inlining.
- Effective elimination of array bounds checks and null pointer checks.
- Efficient tests of type inclusion.

- Architecture-specific code scheduling and register allocation.
- JNI frame improvements to minimize JNI overhead.

For details on how many of these optimizations work, see [1] and [6].

**Selective Compilation**

IBM balances the cost of compilation with the benefit of producing better code via a technology called the Mixed Mode Interpreter, an advanced selective compilation technology. Most of IBM developer kit and runtime environments for Java 2 V1.3 technology include MMI. "Mixed Mode" refers to executing both interpreted and JIT-generated code in the same program. JIT compilation is deferred for certain methods until it is clear that the time spent to compile the methods is justified. Methods not deemed sufficiently important to warrant compilation will continue to be interpreted. The interpreter has also been optimized to execute these methods more efficiently.

To determine when a given method should be JIT compiled, MMI tracks the number of invocations of the method, and the iteration counts of loops in the method, and then uses this data to determine when it is appropriate to invoke the JIT to compile the method. For example, methods that are invoked a small number of times and that do not contain long-running loops will always be interpreted, while methods that are invoked more frequently or that contain long-running loops will be compiled by the JIT. This approach enables fast execution of compute-intensive code, without requiring all methods to incur the overhead of being JIT compiled. In addition, there is a seamless control transfer between interpreted code and JIT-generated code through the use of a C stack. This technique improves performance, and also allows interpreted and JIT-compiled code to share the same exception handling mechanism.

In addition, MMI also performs runtime profiling to obtain information such as which branches are taken for conditional statements, which later is used by the JIT compiler to produce more optimal code. Finally, MMI will trigger JIT compilation either before a method is invoked, or during the execution of a method if the method is running for a relatively long period of time. In the latter case, MMI will dynamically transfer control to the JIT-generated code, rather than waiting until the next invocation of the method. This technique is particularly effective for programs that contain very long running loops in methods that are called infrequently, such as server communication daemons that are called only once but then run for the entire duration of the server program.

**AS/400 Direct Execution Technology**

In addition to JIT, AS/400 also implements a Direct Execution technology that

generates fully compliant, optimized Java applications. It performs interclass binding between the classes within a JAR file, and in some cases, the classes are inlined. Interclass binding improves the call speed while inlining removes the call entirely. In certain cases, it can inline methods between classes within the JAR file or ZIP file. This technology also includes persistent verification.

To prepare class files to run on the AS/400 JVM, OS/400 preprocesses class files using the Java transformer. The Java transformer creates an optimized program object that is persistent and is associated with the class file. In the default case, the program object contains a compiled, 64-bit RISC machine instruction version of the class. The Java interpreter does not interpret the optimized program object at runtime. Instead, it runs when the class file is loaded.

It should be noted that on the AS/400 Java implementation, you can run multiple levels of development kits, including 1.1.6, 1.1.7, 1.1.8, and 1.2.2, on the same JVM.

**Code Base Engineered for Performance**

*Fast Object Allocation*

Typical Java programs allocate objects via *new* or *newinstance*() at a high rate. Efficient and scalable JVM implementations are carefully optimized in this area. IBM object allocation technology builds on the idea of thread local heaps (TLH), introduced by Sun in the 1.1.5 reference platform. TLH allocators allow a thread to satisfy most allocations by splitting a privately held object. Serialization with other threads via locks is required only when the TLH is exhausted or, much less frequently, when a large object is allocated.

The key IBM innovation in this area is the introduction of variable sized thread local heaps. Java heap utilization is greatly increased with this approach as compared with the earlier fixed-size TLH allocator. Free list management is also optimized to favor address ordered object placement, which lowers fragmentation and reduces other overheads, as explained below.

*Streamlined Garbage Collection*

Java virtual machines reclaim memory consumed by unused objects through the process of garbage collection (GC). Since objects cannot be explicitly freed, Java programs require JVMs to implement effective GC strategies.

A common approach to garbage collection alternates periods of allocation by the program with periods of reclaiming unused memory. While garbage is being collected, all threads in the program are stopped. This is sometimes referred to

as "stop the world" GC, and can be contrasted with other approaches to garbage collection which are performed concurrently with program execution.

Naturally, it is desirable for stop the world collectors to reduce the duration of GC pauses, both to lower GC overhead and to minimize disruptions to the program. IBM has developed several innovative techniques which work together to support this goal.

Garbage collection is performed by first *marking* live objects, then *sweeping* the Java heap to discover and coalesce free memory areas. An optional *compaction* step is used to reorganize the heap if it becomes excessively fragmented. In the IBM approach, optimizations are made to marking and sweeping, while measures are taken to avoid costly heap compaction in most cases.

The IBM GC has a highly tuned mark phase which can exploit multiple processors, allowing them to collaborate in the process of marking live objects. This helps the garbage collector scale with the workload on multiprocessor machines, and also reduces pause times in the larger heaps which are typical of this environment.

Free space is reclaimed using a new approach called *Bitwise Sweep*, which scans a dense array of mark bits provided by the mark phase. Since the allocator is TLH based, free areas in the heap smaller than the minimum TLH size are not generally usable. Bitwise Sweep exploits this property by ignoring such small areas and thereby gaining significant speed increase. The result is an extremely efficient process with very low overhead.

As the gap between processor and memory speeds continues to widen, the cost of memory-intensive operations like heap compaction is already very high. Rather than attempt to reverse this trend, the IBM GC and allocator work together to avoid most compactions. The variable sized TLH allocator helps by being flexible about the size of TLH blocks. An address ordered free list yields initial object placements which tend to produce low fragmentation. Garbage collection is triggered a bit early in many cases to preserve a *wilderness area*, which can be used to satisfy a large allocation request when the alternative is to compact the heap.

*Effective Heap Size Management*

Another key IBM innovation helps to correctly size the Java heap for a given application. Rather than simply manage heap size to yield a fixed fraction of free space, IBM implemented JVMs actually measure GC overhead during JVM operation. This allows heap free space to be increased beyond normal targets when allocation rates are especially high. In this way the JVM responds to the actual needs of the program, rather than a simpler but less accurate estimate.

Care is also taken to expand the heap by at least a useful fraction, avoiding the need for frequent expansion events during periods of increasing memory pressure.

When heap utilization is low, memory is returned to the operating system by shrinking the heap. Placing the wilderness area at the end of the heap makes this more effective. Because the wilderness is typically empty, shrinking can often be done without moving any objects.

*Garbage Collector on AS/400 and S/390*

The AS/400 and S/390 Java 2 implementations approach the GC problem slightly differently from the approach described above. These platforms employ a concurrent GC algorithm. This algorithm allows the discovery and collection of unreachable objects without significant pauses in the operation of the Java program. A concurrent collector cooperatively discovers the references to objects under the running threads, instead of a single thread.

As discussed above, many garbage collectors are termed "stop-the-world". This means that at the point where a collection cycle occurs, all Java worker threads stop working and garbage is completely collected. On many JVM implementations this process is single threaded (as noted above IBM has overcome this limitation). During the collection, Java programs experience a pause while the collector does its work. The AS/400 and S/390 algorithm never stops the program threads. It allows those threads to continue operation while the garbage collector completes its task. This prevents pauses, and allows all processors to be used during garbage collection.

*Efficient Synchronization*

The Java programming language provides monitors which allow users to synchronize multiple threads access to shared data. Synchronization is enforced at the object level and is employed by either invoking a synchronized method or encapsulating a critical code section inside a synchronized block. Since object oriented code design tends to produce many small methods, and it is often the case that these methods are syncrhonized, the performance of the synchronization package provided by the JVM is crucial to overall system performance. Specifically, the time required to acquire and release a semaphore which is free or uncontended is especially important because the majority of monitor acquisitions are successful on their first attempt (e.g., there is no contention).

Given the importance of monitor performance, IBM has invested heavily in researching and developing the highest performing implementation. The result is the platform independent thin lock implementation detailed in the paper by

Bacon, et. al., [4] and the deflation enhancement detailed by Onodera and Kawachiya [5]. The lock design takes advantage of a new object layout to embed a featherweight thin lock (24 bits) in the header of each object. The high bit of this thin lock delineates between two basic lock states: inflated and flat. For the flat lock the 24 bits consist of an indicator as to lock type (flat or inflated), a thread identifier, and a recursion count. In the inflated case the bits contain the lock type indicator plus a pointer to a backing inflated lock struture.

The structure of the flat lock allows an extremely efficient acquire/release path. When there isn't any contention (the common, high performance case) the lock can be acquired in a few instructions by atomically swapping the current thread id with the thin lock. Releasing requires nothing more than zeroing the appropriate bits in the header. The Inflated locks are required if contention for the monitor has occurred and a mechanism to block a thread on an event is needed. The acquisition of the contended lock has also been fine tuned in the IBM implementation for high performance. A three tier spin-then-yield loop has been implemented which minimizes excessive context switching on highly contended locks with short hold times.

Monitor implementations in previous IBM JVMs inflated a monitor on the first occasion of contention. IBMs Java 2 implementations have significantly improved this design by allowing locks to deflate once a period of contention is over. This allows the high performing thin lock acquire to occur more frequently. This new innovation clearly puts IBM at the forefront of Java monitor technology.

*Scalability*

Given IBM's experience in large, scalable systems, it is not surprising that the IBM Java 2 implementation provides the most scalable JVM in the marketplace. When evaluating IBM Java 2 implementations for speedup as more processors are added to a system it is not uncommon to see server workloads scaling with better than 85% efficiency. This is especially important given IBM's wide range of server hardware.

Scalability in the IBM implementations was ensured through extensive analysis of representative workloads. The workloads employed consisted of multithreaded database, Websphere (servlets and JSPs), and business logic applications. It was determined early in the implementation cycle that access to the Java heap could become an impediment to scaling. To avoid this the memory subsystem was restructured to avoid long hold times of locks protecting the data and meta-data pertaining to memory management. It was also determined that server applications which consist of large proportions of native code, specifically type 2 JDBC drivers, did not scale due to impediments in the JNI structures of the JVM. To address this, the JNI subsystem was re-implemented to provide nearly linear scalability in the latest IBM releases.

Significant effort has also been expended to allow the class loader and reflection components of the JVM scale efficiently. Finally, where possible, the class libraries were enhanced to reduce over-synchronization. Please note that this last piece of work was accomplished while maintaining strict Java 2 compliance. This ensures that Java workloads will not only function properly across a range of IBM systems, but scale admirably.

The result of this work is most clearly evident in large, multithreaded workloads. For instance, IBM continues to boast the highest Volano scores in the industry on symmetrical multiprocessing machines. In addition, internal IBM benchmarks stressing web serving and database access show impressive scaling.

*Fast JNI*

Unlike Java 1.1 which used the Native Method Interface (NMI), the Java2 class libraries use the java Native Interface (JNI) to invoke native methods. JNI imposes additional overhead and it significantly reduces performance of frequently used paths, especially in graphics and I/O.

Several enhancements were implemented to provide a fast JNI on the Java 2 platform from IBM. These include the reuse of Java stack frames to reduce JNI method invocation overhead, speeding up key JNI primitives, and tuning class library methods to avoid unnecessary data copying and to use JNI more efficiently in socket read/write, file read/write, and graphics primitives.

*Efficient Networking*

IBM's Java 2 implementations improve Java network performance by reducing the overhead of the Java sockets API. The following improvements were made:

- More efficient use of JNI.
- Reduced transitions across Java-native boundary.
- Efficient mapping to native sockets.

*IBM's Implementations of Java 2 Security*

The Java 2 security model provides policy-based, easily configurable, fine-grained access control. Its implementation creates additional JVM overhead. For example, security checks require file names in canonical form to insure correct comparison. However, conversion to canonical form can be expensive on some platforms. To alleviate this problem, IBM's Java 2 V1.2 implementations avoid computation of canonical form until necessary and cache canonical form translations for reuse.

**Performance Results: Windows, Linux, AIX, and OS/400**

As a result of the extensive work highlighted in this paper, IBM's implementations of Java 2 technology have driven Java performance to an unprecedented level. While a single industry standard benchmark which can successfully quantify the range of IBM JVM performance improvements, a few well accepted benchmarks-- SPECjvm98, VolanoMark, and SPECjbb2000 -- will provide an indication of some of the gains. As always, the JVM user is encouraged to benchmark JVM implementations against their own applications.

## SPECjvm98 Benchmark

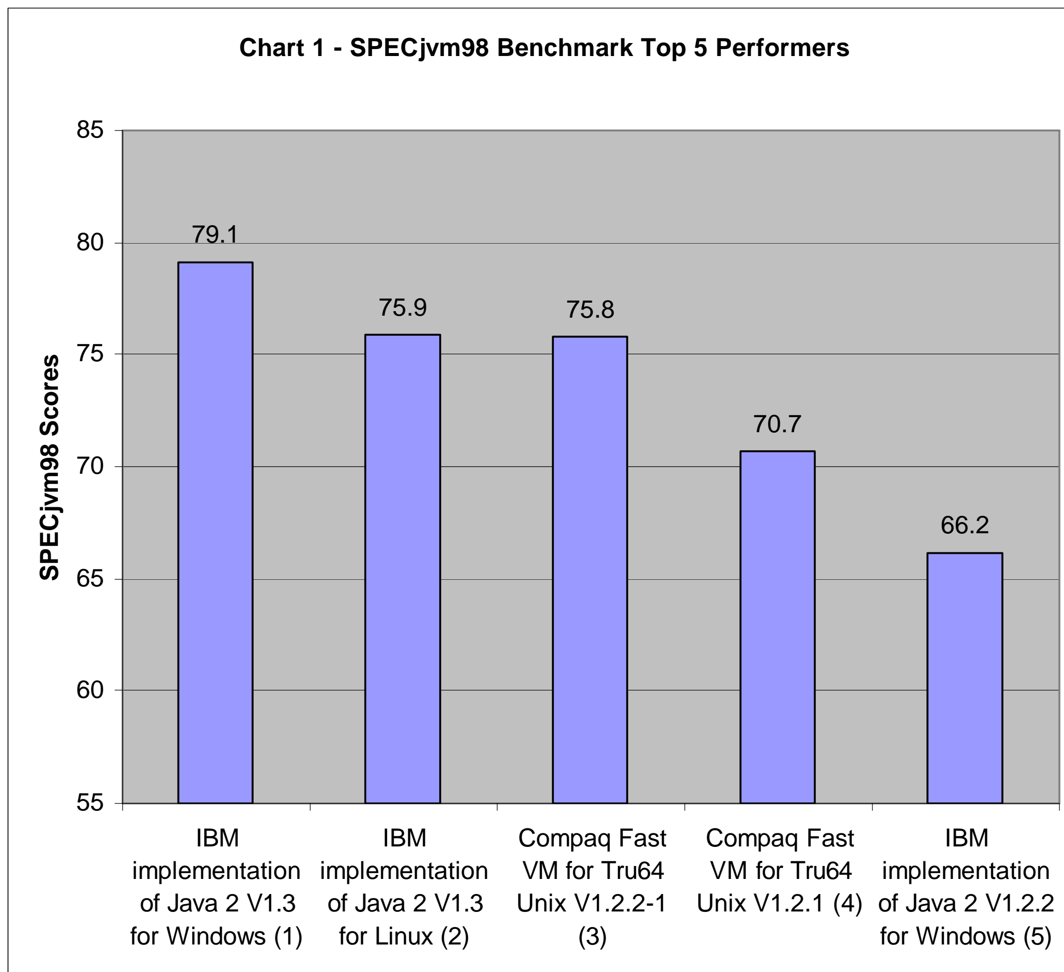
SPECjvm98 is a well accepted industry standard benchmark produced by the non-profit SPEC consortium, *http://www.spec.org*. IBM, as well as all other major computer vendors who provide JVMs, is a member of the SPEC consortium and endorse the SPECjvm98 benchmark. SPEC has produced many of computing industry's most widely used performance benchmarks, and therefore it is not surprising that the SPECjvm98 benchmark has emerged as one of the key Java industry benchmarks.

The benchmark comprises seven tests all of which are computationally intensive. The tests are for the most part single threaded so scalability is not a major component of the score. Graphics performance is also not relevant to the final score. Two of the tests do significant file I/O, but for the most part the benchmark tests the quality of the code produced by the JVM's JIT, the JVM's memory subsystem, and the infrastructure of the virtual machine.

### *Highest SPECjvm98 Benchmark Results*

IBM's implementations of Java 2 technology have driven Java performance to an unprecedented level. Chart 1 lists the top 5 performers in the 48-256 MB category based on results submitted to SPEC. The values shown are the scores reported by the benchmark when running in the SPEC compliant mode. Higher scores mean better performance. The highest score is 79.1 reported by IBM on its Java 2 V1.3 implementation for Windows platform followed by the score of 75.9 obtained with the IBM's implementation of Java 2 V1.3 for Linux platforms.

It is interesting to note that the highest non-IBM Intel-based score is 31.3 obtained with Sun Java 2 SDK w/HotSpot 1.0, Windows NT 4.0, Java Web Server 1.1.3, tested on a Dell Precision Workstation 610 2-450 MHz.




For a description of SPEC, visit http://www.spec.org. To access all submitted results, from SPEC main page, follow *SPEC JVM98 -> Submitted Results -> All JVM98 Results*.

***NOTES:***

*(1)* The SPECjvm98 score of 79.1 was obtained with IBM's implementation of Java 2 V1.3 for Windows, Build cn130-20000313, Microsoft Windows NT 4.0 Server, SP6, Microsoft IIS 4.0, Dell XPS B866r 866 MHz Pentium III, 256 KB L2 cache, 256 MB memory.

*(2)* The SPECjvm98 score of 75.9 was obtained with IBM's implementation of Java 2 V1.3 for Linux, Build cxdev-20000425, Red Hat Linux 6.1, Apache Web Server 1.3.9-4, Dell XPS B866r 866 MHz Pentium III, 256KB L2 cache, 256 MB memory.

*(3)* The SPECjvm98 score of 75.8 was obtained with Compaq Fast VM for Tru64 Unix V1.2.2-1, Java 1.2.2-3, Digital Unix 4.0F, Zeus Web Server 3.3.5, Alpha Server DS 20 6/667, 8 MB L2 cache, 256 MB memory.

*(4)* The SPECjvm98 score of 70.7 was obtained with Compaq Fast VM for Tru64 Unix V1.2.1 beta1, Java 1.2.1-2, Compaq Tru64 Unix 4.0F, Zeus Web Server 1.3.0, Professional Workstation XP 1000 6/667, 4 MB L2 cache, 256 MB memory.

*(5)* The SPECjvm98 score of 66.2 was obtained with IBM's implementation of Java 2 V1.2.2 for Windows, Build cn122-20000127, Microsoft Windows 2000 Advanced Server, Microsoft IIS 5.0, IBM Netfinity 5600 800 MHz Pentium III, 256 KB L2 cache, 256 MB memory.
*(6)* Competitors' results are provided for comparison. All competitive results shown are based on the benchmark measurements conducted by the respective companies. IBM did not test or in any way verify the results obtained by these companies. The configuration of the system under test as well as the test environment may vary. Readers are encouraged to examine the companies' published disclosure reports for details concerning the system configuration and the methodology used to obtain the published results.

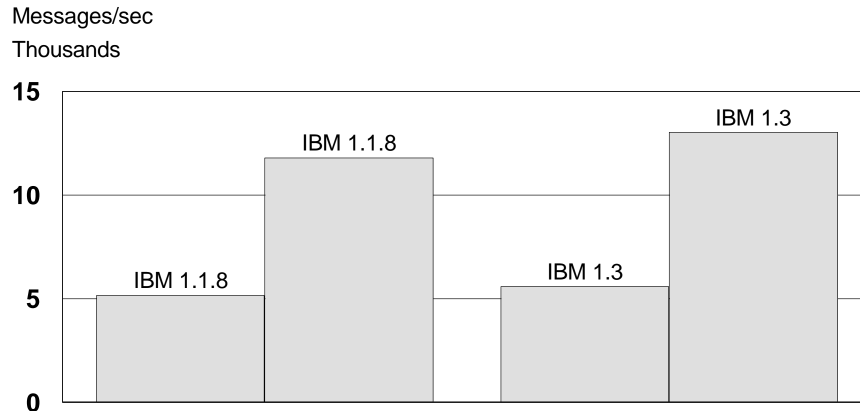
### VolanoMark 2.1.2 Benchmark

VolanoMark 2.1.2 is a widely used industry benchmark, produced by Volano LLC. It was one of the first Java server benchmarks to be distributed, and has gained acceptance in the industry as an important server benchmark. It is easy to setup, runs quickly, and has been used in press articles (JavaWorld) and vendor announcements as a measure of Java server performance. These factors all contribute to make VolanoMark a key benchmark to analyze and track.

The benchmark can be run in 2 modes: loopback (i.e., local) and network. The result of the loopback test represents throughput in terms of number of messages per second. Higher numbers mean better performance.

Chart 2 shows the results of the loopback test on a 1- and 4-way configuration running under Windows NT 4.0 with IBM's implementations of Java 1.1.8 and 1.3. On a 1-way configuration, IBM's implementation of Java 2 V1.3 is 8% faster than IBM's implementation of Java 1.1.8. On a 4-way configuration, the performance improvement is 10%. In terms of processor scalability, going from 1-way to 4-way configuration, the benchmark performance improves 2.28 times on IBM's implementation of Java 1.1.8 but 2.33 times better with Java 2 V1.3.

**Chart 2 - VolanoMark 2.1.2 Loopback Test - Windows**




| | IBM 1.1.8 | IBM 1.3 |
|---|---|---|
| 1-way | 5164 | 5587 |
| 4-way | 11814 | 13020 |

***NOTES:***

*(1)* IBM 1.1.8 for NT: IBM's implementation of Java 1.1.8 for Windows, IBM Build 118-0728, Microsoft NT 4.0 Enterprise Edition with SP 4.
*(2)* IBM 1.3 for NT: IBM's implementation of Java 2 V1.3 for Windows, IBM Build cndev-20000424, Microsoft NT 4.0 Enterprise with SP 4.

All tests were run on an Intel Pentium III 550 MHz, 2 MB L2 cache, 4 GB of memory, using the Volano benchmark run rules.

On a 1-way configuration using the same hardware described above, the loopback test scores for IBM's implementations of Java V1.1.8 and V1.3 for Linux platform were 5177 and 6647, representing 28% improvement. The test was run under Red Hat Linux 6.1.

**AIX outstanding VolanoMark score**

In independent Java performance and scalability testing, an 8-way RS/6000 M80 set outstanding performance, attaining two-and-a-half times the performance of the previous record holder, a 22-way E6500 server from Sun Microsystems.

On the VolanoMark network scalability test, an 8-way RS/6000 M80 with 4GB of memory running Java 2 version 1.2.2 and AIX 4.3.3 trounced a 22-way Sun E6500 with 30GB of memory. With 9,000 connections, the M80 transferred 11,960 messages per second, 147 percent faster than the Sun E6500's 4,847

messages per second. The M80 further extended its dominance in the Java space by becoming the only system to successfully test 11,000 connections on VolanoMark, transferring 7,175 messages per second.

In VolanoMark 2.1.2 local performance testing with 200 connections, an 8-way RS/6000 M80 transferred 46,370 messages per second, an 84 percent increase over the 25,131 connections posted by a Sun E6500 server configured with 22 processors.

For more details, visit the RS/6000 home page, *http://www.rs6000.ibm.com,* and *http://www.rs6000.ibm.com/resource/pressreleases/2000/May/m80_records.html*

**IBM AS/400e server sets new VolanoMark record**

IBM's AS/400e Model 840 server is the first to achieve a six-digit score on Volano LLC's VolanoMark 2.1.2 local performance test, surpassing the results posted by a comparable Sun Microsystems server by a factor of four. The AS/400e Model 840 handled a record 108,153 messages per second, using 200 concurrent connections.

In independent testing, the 24-way AS/400e Model 840 with 4GB of memory was more than 300 percent faster than Sun's E6500 22-way server with 30GB of memory. The AS/400 Model 840-2420 was powered by IBM's I-Star (copper and silicon-on-insulator) processors and running IBM's implementation of Java 2 version 1.3 on OS/400 V4R5.

Version 4 Release 5 of OS/400 includes enhancements that deliver up to 65 percent improvement in Java performance over the previous release of OS/400.

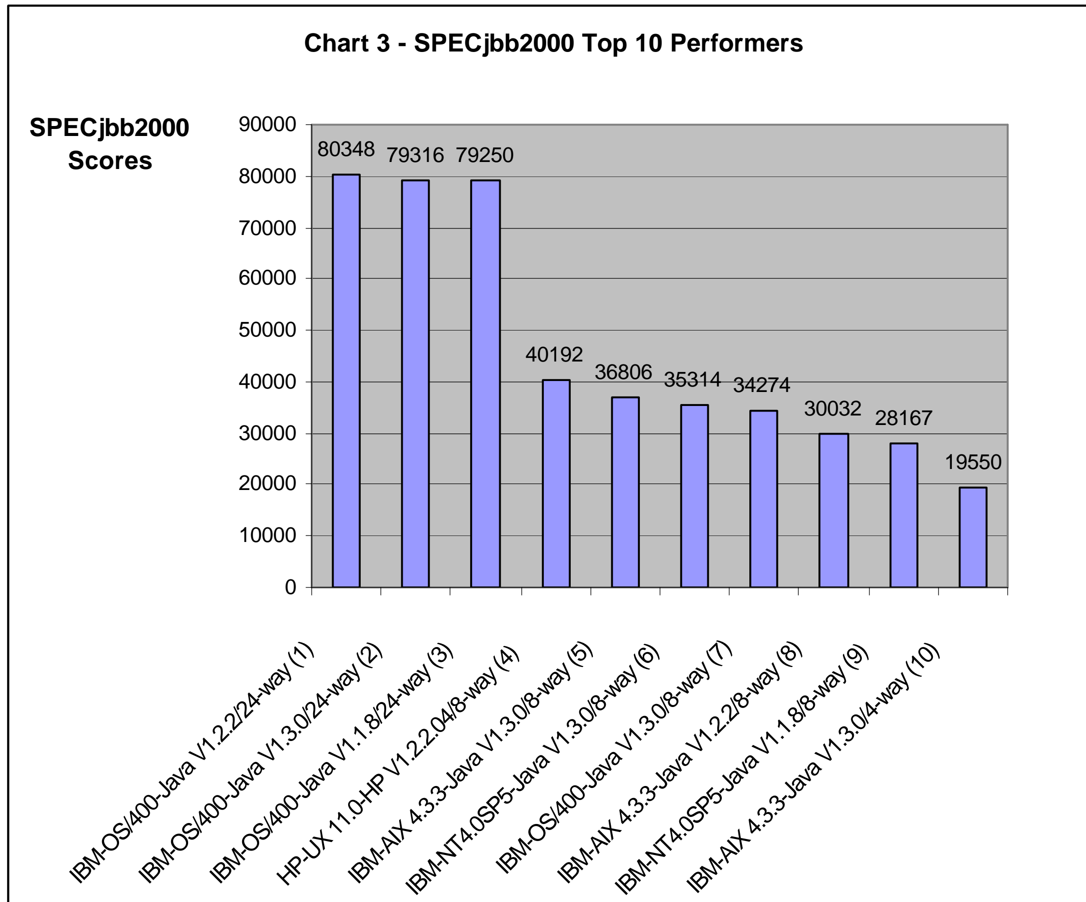
For more details, visit the AS/400 home page, *http://www.as400.ibm.com*.

**SPECjbb2000 Benchmark**

SPECjbb2000 is a new SPEC benchmark that measures the implementation of JVM, JIT, garbage collection, threads and some aspects of the operating system. It also measures the performance of CPUs, caches, memory hierarchy and the scalability of Shared Memory Processors (SMPs) platforms on the specified commercial workload. The benchmark does not measure AWT, network, I/O, and graphics performance.

The score generated by SPECjbb2000 is ops/second. It is a composite throughput measurement representing the averaged throughput over a range of measurement points. It is described in detail in the document *"SPECjbb2000 Run and Reporting Rules.* Higher scores mean better performance.

For a detailed description of SPECjbb2000, visit *http://www.spec.org*. To read the benchmark documentation, click on the *SPEC JBB2000* url *http://www.spec.org/osg/jbb2000/*. To access all submitted results, from SPEC main page, follow the links from *http://www.spec.org/*. Chart 3 illustrates the top 10 performers where the Java 2 implementation by IBM on OS/400 platform records the highest score followed by Java 1.1.8 implementation also from IBM.




***NOTES:***

*(1)* The SPECjbb2000 score of 80348 was obtained with IBM's implementation of Java 2 V1.2.2 for OS/400 V4R5M0, running on AS/400e 840-2420 24-way 500 MHz, 256 KB L1 and 8192 KB L2 caches, 16000 MB memory.

*(2)* The SPECjbb2000 score of 79316 was obtained with IBM's implementation of Java 2 V1.3.0 for OS/400 V4R5M0, running on AS/400e 840-2420 24-way 500 MHz, 256 KB L1 and 8192 KB L2 caches, 16000 MB memory.

*(3)* The SPECjbb2000 score of 79250 was obtained with IBM's implementation of Java V1.1.8 for OS/400 V4R5M0, running on AS/400e 840-2420 24-way 500 MHz, 256 KB L1 and 8192 KB L2 caches, 16000 MB memory.

*(4)* The SPECjbb2000 score of 40192 was obtained with HP's implementation of Java 2 V1.2.2.04 for HP-UX 11.0, running on N4000 8-way 552 MHz, 1536 KB L1 cache, 4096 MB memory.

*(5)* The SPECjbb2000 score of 36806 was obtained with IBM's implementation of Java 2 V1.3.0 for AIX 4.3.3, running on RS/6000 7026-M80 8-way 500 MHz, 256 KB L1 and 4096 KB L2

caches, 4096 MB memory.
*(6)* The SPECjbb2000 score of 35314 was obtained with IBM's implementation of Java 2 V1.3.0 for Windows, NT 4.0 with SP5, running on IBM Netfinity 8500R 8-way 700 MHz, 32 KB L1 and 2048 KB L2 caches, 4096 MB memory.
*(7)* The SPECjbb2000 score of 34274 was obtained with IBM's implementation of Java 2 V1.3.0 for OS/400 V4R5M0, running on AS/400e 840-2403 8-way 540 MHz, 256 KB L1 and 4096 KB L2 caches, 8000 MB memory.
*(8)* The SPECjbb2000 score of 30032 was obtained with IBM's implementation of Java 2 V1.2.2 for AIX 4.3.3, running on RS/6000 7026-M80 8-way 500 MHz, 256 KB L1 and 4096 KB L2 caches, 4096 MB memory.
*(9)* The SPECjbb2000 score of 28167 was obtained with IBM's implementation of Java V1.1.8 for Windows, NT 4.0 with SP5, running on IBM Netfinity 8500R 8-way 700 MHz, 32 KB L1 and 2048 KB L2 caches, 4096 MB memory.
*(10)* The SPECjbb2000 score of 19550 was obtained with IBM's implementation of Java 2 V1.3.0 for AIX 4.3.3, running on RS/6000 7026-M80 4-way 500 MHz, 256 KB L1 and 4096 KB L2 caches, 4096 MB memory.

## Conclusion

IBM has continued its systematic enhancements to Java performance technology. Measurement data on implementations to ship this year demonstrate the significant performance benefit of this technology and IBM's commitment to deliver open standards technology for e-business.

## References


[1] *Overview of the IBM Java Just-In-time Compiler*, Suganuma et al., IBM Systems Journal, Vol. 39, No. 1, 2000.

[2] *The Evolution of a High-Performing JVM*, Gu et al., IBM Systems Journal, Vol. 39, No. 1, 2000.

[3] *Java Server Performance: A case study of building efficient, scalable JVMs,* Dimpsey, et al., IBM Systems Journal, Vol. 39, No. 1, 2000.

[4] *Thin Locks: Featherweight Synchronization for Java*, David Bacon et al. Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, 1998, 258-268.

[5] *A Study of Locking Objects with Bimodal Fields*, Tamiya Onodera and Kiyokuni Kawachiya. OOPSLA'99 Conference Proceedings, 1999, 223-237.

[6] *Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler*, K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani, ACM SIGPLAN JavaGrande Conference, June 1999.


## Disclaimer

The benchmarks and values shown here were derived using particular, well-configured computer systems. All performance benchmark values are provided "AS IS" and no warranties or guarantees are expressed or implied by IBM. Actual system performance may vary and is dependent upon many factors including system hardware configuration, software design and configuration. Buyers should consult other sources of information to evaluate the performance of systems they are considering buying and should consider conducting application-oriented testing. For additional information about benchmarks, values and systems tested, please contact your IBM local Branch Office or IBM Authorized Reseller.

## Notice and Trademarks

### Notice

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's products, programs, or services may be used.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to those patents. License inquiries can be sent, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

### Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX AS/400 IBM Netfinity OS/2 S/390 OS/390

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark in the United States and other countries and is licensed exclusively through The Open Group.