# WHITE PAPER

August, 1997

Prepared By
Internet Business Unit

Compaq Computer

## CONTENTS

# Java Performance on Compaq's ProLiant Servers

*Java Performance has been a major debate in the Internet/intranet industry ever since Java was introduced to the computing world. The initial JVM (Java Virtual Machine) and browser product offerings interpreted Java applets, and the performance was found to be significantly slower than C and C++ programs. Then JIT (Just In Time) Compilers, which compile the byte code on a function by function basis on-the-fly, were introduced. This significantly improved the applet performance in many areas. JIT Compilers vastly improved the performance of most non-graphical benchmarks over the interpreters but did not significantly improve the performance for graphical benchmarks.*

*This White Paper describes the results of Java benchmark tests run on Compaq's ProLiant 800 servers and Sun Ultra 2 servers. The ProLiant 800 outperformed Sun Ultra2 by an impressive 250% in price and 235% in performance. Detailed test results are provided in Appendix A, "Details of Caffeine Mark Test Results" and Appendix B, "Details of JMark Test Suite Results".*

COMPAQ

## NOTICE

The information in this publication is subject to change without notice.

**COMPAQ COMPUTER CORPORATION SHALL NOT BE LIABLE FOR TECHNICAL OR EDITORIAL ERRORS OR OMISSIONS CONTAINED HEREIN, NOR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL.**

This publication does not constitute an endorsement of the product or products that were tested. The configuration or configurations tested or described may or may not be the only available solution. This test is not a determination of product quality or correctness, nor does it ensure compliance with any federal, state or local requirements. Compaq does not warrant products other than its own strictly as stated in Compaq product warranties.

Java Performance on Compaq's ProLiant Servers

First Edition (August 1997)
Document Number ECG039.0897

## INTRODUCTION

Java applet or application performance depends on many aspects of the Java Virtual Machine and the associated packages and class libraries. Even before Java came on to the computing horizon, interpreters were almost always slower than compilers. Compilers, (excluding non-JIT compilers) have the luxury of performing extensive optimizations on the source code. They perform extensive dataflow and control flow analysis on an intermediate representation of the original source code and produce a highly optimized hardware specific code. All of this processing is being done "ahead of time" or at compile time, not at runtime in the traditional languages like C and C++. JIT Compilers are a compromise between the low performing interpreters and high performing non-dynamic, non-real-time, "ahead of time" traditional compilers. JIT Compilers compile the Java byte code on-the-fly when an applet is running in the browser and they compile a function at a time. This has advantages and disadvantages. Because of their real-time compilation of the code function at a time, they do not have the luxury of doing extensive optimizations like C and C++ compilers do, especially inter-procedural (across the functions or methods) optimizations; however they compile only those functions or methods that are called. JITs are quickly catching up with the traditional compilers, but at the time of writing this white paper they still lag the C and C++ compilers in performance.

Among all the VM components, the interpreter, JIT compiler and the security components affect the applet performance more than any other components. An applet performance is directly dependent on how fast the interpreter interprets the code on the native system and to the extent it is security checked by the security components of the VM. With JIT compilers, it also depends on the optimizations performed by the JITs on the byte code. Extensive optimizations by the JIT compilers could result in efficient machine code, which reduces the runtime of the applet but adds to the compile time of the applet.

## EXECUTIVE SUMMARY

Java is a language and a runtime system designed for Internet/intranet applications. The single most important component of the Java system is the Java Virtual Machine (JVM). It is the JVM that enables Java applets to be platform independent and architecturally neutral. The JVM is a stack based "software machine" or "soft CPU" that has a set of instructions, data types, a set of registers, a stack, a garbage collection heap and a method area. All these are logical abstract components of the JVM. It supports 248 byte codes, each performing a basic CPU operation such as adding an integer to a register, jumping to subroutines, storing a result in memory, and incrementing or de-incrementing a register. The VM is, in effect, a stacked ALU with local and global variables.

Java is considered an interpreted language. This is not entirely true. The Java source language is compiled into an intermediate form known as byte codes by the javac compiler. This intermediate language (IL) or the byte codes are JVM instructions and data. In non-java compilers the IL is an abstraction of hardware for which the compiler is written. These compilers optimize the IL and generate code for the specific hardware in the backend of the compiler which is then linked with libraries, loaded and executed. Java JIT (Just In Time) Compilers on the other hand optimize the byte code and produce code for the native hardware and execute it at the same time. The compile, link, load and execute operations are separate and discrete on non-java compilers whereas it is a single continuous dynamic on-the-fly operation in Java interpreters or JIT compilers. See Figure 1 for an overview of Java applet processing (i.e. compile, load/download, interpret or JIT compile and run cycle).

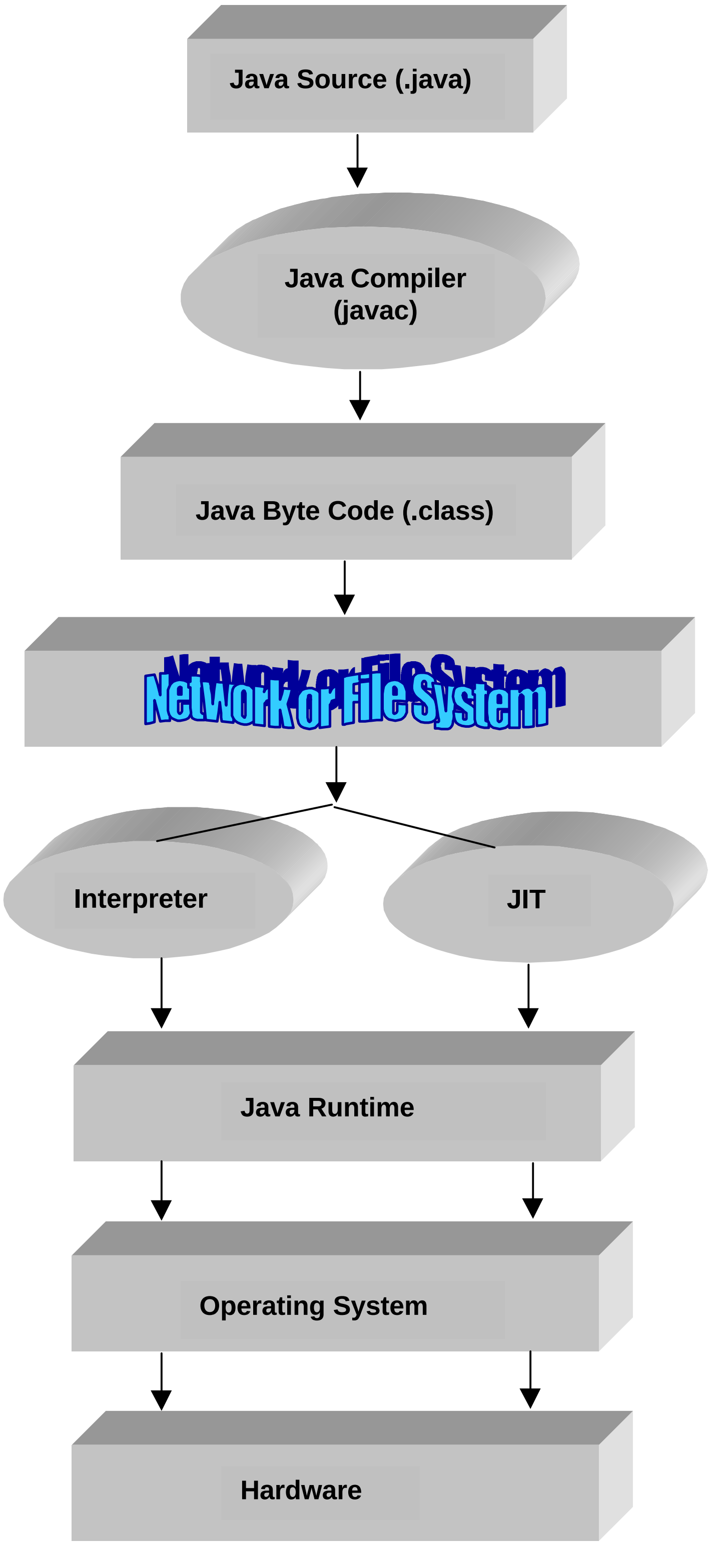


*Figure 1 – Overview of Java applet processing*

Java's performance over the last 2 years has been disappointing. Java applets still do not run as fast as C, or C++ applications. Java security has a profound effect on JVM performance. For instance, on some JMark 1.0 tests run by PC magazine, Borland's 5.0 JIT Compiler was 50 to 60% faster than Netscape's JIT compiler, despite the fact that Netscape licensed the same JIT Compiler from Boreland. The reason for this degradation in performance is Netscape's very aggressive "Sandbox" security and byte code verifier.

Asymetrix SuperCede, one of the top 3 VMs in the industry, bypasses the security checks made during the runtime of an applet. It scored very high on many benchmark tests not only because of its "flash" compiler technology but also because of bypassing the VM security. SuperCede's results are included in this paper’s charts, mainly to illustrate the performance enhancement a VM can gain by bypassing the extreme security requirements of Java security.

The fundamental problem with Java Security is that it is a run-time/real-time security system. The sandboxing, byte code verification, and security manager checks are all done in real-time when the applet is running within the browser in front of a user.

## Test Objectives

The Intranet and Groupware department at Compaq ran Java benchmark tests on Compaq's ProLiant 800 Servers and Sun Ultra 2 Servers to investigate three major questions:

- How do Compaq's servers perform against Sun Microsystems in the Java space?
- Which VMs perform the best on Compaq servers?
- How do the JITs perform in comparison to the interpreters on Compaq servers?

Three major benchmark test suites for the Java industry (CaffeineMark, JMark and Linpack) were selected to run on the Compaq ProLiant 800 servers and Sun Ultra 2 machines. The following JVMs were benchmarked:

- Microsoft's VM (MSVM) using Microsoft Internet Explorer (MSIE) 3.0
- Symantec's Visual Café (VCafe) 1.0 applet viewer
- Asymetrix's SuperCede (SC) 1.0 applet viewer
- Netscape's VM (Nav) in its Navigator Browser 3.0
- SUN Java Developer Kit (JDK) 1.0.2

**NOTE:**
*SUN has recently released JDK 1.1, which is the latest release, however at the time of these benchmark trials, only JDK 1.0.2 had a JIT compiler.*

CaffeineMark tests suite and Linpack ran successfully on both ProLiant 800 systems and Sun Ultra 2 systems. JMark 1.0 test suite could not be run on the Sun server.

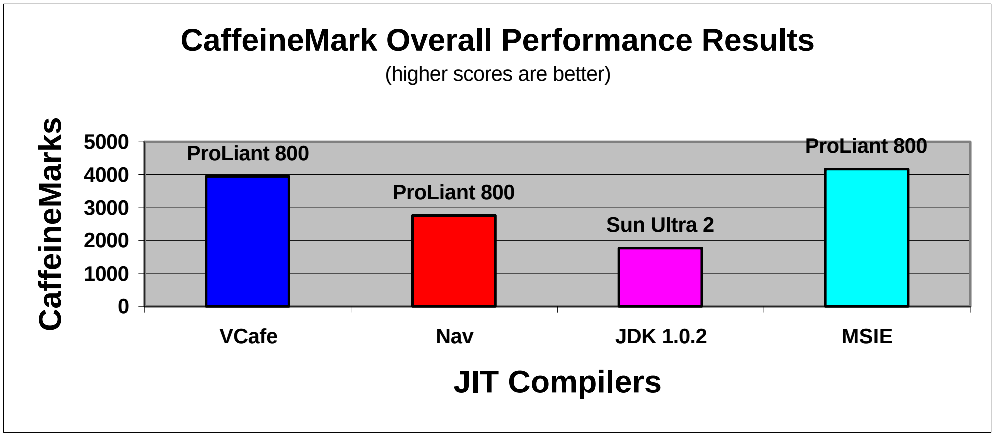
The ProLiant 800 had 32M memory with a 200Mhz CPU and 256k cache. The Sun Ultra 2 had 256M memory, two 167Mhz CPUs, 256k cache. Each test was run three times to get an average score.

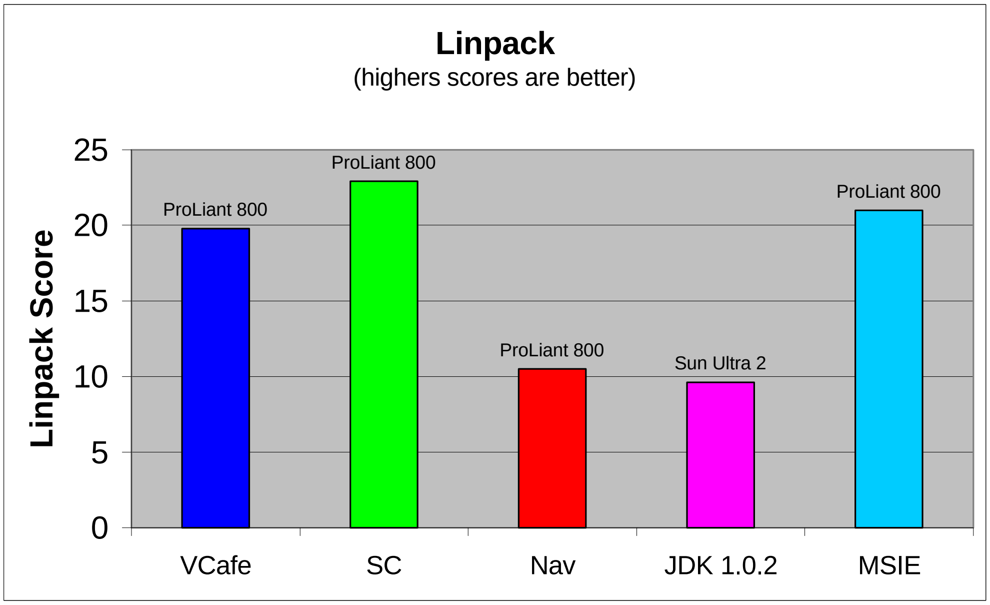
## Summary of the Test Results

***NOTE:***
*The ProLiant 800 outperformed Sun Ultra2 by an impressive 250% in price (the ProLiant 800 costs $8000 vs. Sun Ultra 2 systems costing $20,000) and 235% in performance.*

- Compaq's ProLiant 800 server outperformed Sun Ultra2 system in almost all the benchmarks by an impressive margin, especially in price and performance (Microsoft's VM Just In Time Compiler on ProLiant 800 is 235% faster than Sun Ultra 2's JDK 1.0.2 VM JIT compiler). See the overall CaffeineMark chart and Linpack chart for a quick overview of the results and individual test result charts for details.
- MSVM (Microsoft's Virtual Machine) running in MSIE 3.0 (Microsoft's Internet Explorer) browser or the VJ++ IDE is the fastest VM in the Java Space currently.
- The top 3 VMs in the Java Space are MSVM, Symantec's Visual Café and Asymetrix's SuperCede.
- Slowest JIT on ProLiant 800, Netscape Navigator 3.0, is 156% faster than Sun's JDK 1.0.2 JIT using the CaffeineMark test suite.
- Fastest JIT on ProLiant 800, Microsoft's MSIE 3.0 VM, is 235% faster than Sun's JDK 1.0.2 running the CaffeineMark test suite.
- Compaq's ProLiant 800 running MSIE 3.0 VM (JIT) is 218% faster than Sun's JDK 1.0.2 JIT on the compute intensive Linpack suite.
- MSIE JIT is 18 to 100 times faster than MSIE interpreter on the ProLiant systems on the non-graphics tests. On the graphics tests the JITs do not show any significant improvements in speed.
- Sun Ultra 2's JDK 1.0.2 JIT is 3 to 50 times faster than its interpreter on non-graphics tests. On the graphics tests, again the JIT does not show any significant improvement over the interpreter.
- JDK 1.1 interpreter on Sun Ultra 2 platform is on an average twice as fast as JDK 1.0.2 interpreter on the same platform.
- JDK 1.1 interpreter on the ProLiant platform is twice as fast as JDK 1.1 interpreter on the Sun Ultra 2 platform on all non-graphics tests. On the graphics tests, Sun's Ultra 2 JDK 1.1 interpreter is 2 to 3 times faster than the interpreter on ProLiant.

## Summary Charts of CaffeineMark and Linpack Test Suite





## ANALYTICAL SUMMARY OF THE TEST RESULTS

What was learned from the CaffeineMark test suite is that the MSIE 3.0 (Microsoft Internet Explorer) or VJ++ 1.0 (Visual J++ IDE) JIT compiler is currently the fastest JIT in the Java space. Asymetrix SuperCede scored very high on many tests due to it bypassing the virtual machines' security code. All other VMs do not in a real life environment, therefore Asymetrix's results were discounted as a reflection of true performance of a VM in a real life Java environment.

The JIT compilers show significant performance improvements over interpreters on all the non-graphical or math heavy/compute intensive tests. On an average, MSIE 3.0 VM (JIT) on the ProLiant 800 is 2 to 10 times faster than Sun's JDK 1.0.2 VM (JIT) on the Ultra 2 system. On the graphics tests, there were some interesting results. The JIT Compilers do not increase the performance of the VMs as well as on the compute intensive tests. This is mainly because of the poor implementation of the AWT package of the Java VM. Most graphics operations are implemented in the native code (X Windows on Solaris platform and Win32 subsystem on NT), consequently the JITs could not optimize the graphics byte code to the extent they could with math tests. In fact the interpreters performed much better than the JITs on these tests, indicating that the JITs may sometimes degrade performance.

*Please note that most server applications and servlets are non-graphical in nature. It is the optimization of compute intensive and I/O intensive (for database and network applications) operations that are more important for server performance.*

Linpack is a compute intensive test suite originally designed for super computers. MSVM, Symantec Visual Café and SuperCede have all performed equally well and are twice as fast as Sun's JDK 1.0.2 VM. JIT Compilers speedup performance by a factor of 20 over the interpreters.

One interesting result to note here is the performance of Sun's JDK 1.1 interpreter on the ProLiant 800 and JDK 1.1 interpreter on the Sun Ultra 2 system. On almost all CaffeineMark and Linpack tests, Sun's interpreter performed better on the ProLiant platform in comparison to Sun's own Ultra platform. This indicates that the underlying OS and HW (NT and INTEL) are responsible for the increased performance over Sun's OS and HW (Solaris and SPARC).

JMark test suite confirmed our earlier findings from CaffeineMark and Linpack. JIT Compilers have performed significantly better than interpreters on the compute intensive or non-graphics tests. Interpreters have done as well as the JITs and better on some of the graphics tests.

Java security has a profound affect on Java performance. This is confirmed by the test results. Asymetrix's SuperCede does not do security checks while running the applets like all other VMs that were tested. Its performance numbers were excellent. SuperCede derives its better performance due to two reasons:

- It does extensive optimizations of the byte code much like the JITs from Microsoft or Symantec's Visual Café.
- It bypasses the runtime security checks of the byte code and other VMs that were tested do not.

So in all the test case results in this white paper, Asymetrix SuperCede's results should be adjusted by a certain percentage before it can be compared against other VMs or JITs fairly. Sun's JIT has performed the poorest on almost all of the tests.

## JAVA SECURITY AND PERFORMANCE

As mentioned earlier, an applet performance is dependent not only on the interpreter and the JIT compiler performance, but also on the efficiency of the VM security components. For instance, on JMark l.0 test suite, Borland's 5.0 JIT had a score of 5711 JMarks. Netscape, which licensed Borland's JIT scored 2138, about 50 to 60% less than Borland's JIT. The reason for this degradation of performance, despite the fact that the JIT is the same in both VMs, is Netscape's very aggressive "sandbox" security and byte code verifier. Additional details follow regarding the Java security components: sandbox, byte code verifier, class loader and security manager. A simple mathematical relationship between JVM performance and its components is as follows:

JVM Performance ✪ $\frac{1}{A + B + C + D}$

A = Execution time of JIT produced code
B = Execution time of JVM packages
C = JIT compile time
D = Execution time of the security components

Asymetrix SuperCede, one of the JVMs we tested, is one of the top three VMs in the Java Space currently. It performs exceptionally well on many tests not only because of its "Flash" compiler technology (which performs extensive optimizations of the byte code), but also because it bypasses the security checks in its VM implementation. A brief description of Java Security Architecture is given below to explain the Java Performance and Java Security relationship.

## Java Security Architecture

From the beginning, Java architects have placed emphasis on Java security at the expense of performance. There are two major aspects of Java technology that affect Java's performance:

- It is an interpreted system and interpreted systems are almost always slower than compiled systems.
- Java's runtime security that does extensive checks of not only the byte code but also runtime resource access validation of applets.

Java Security is architected, designed and implemented into 3 major components of the Java System:

1 Java Language

2 Compiler

3 Runtime Mechanisms

### Language and Compiler Security Features

First and the foremost is the removal of pointer based operations from Java Language. Unlike in C or C++, in Java, all accesses to memory areas must be done using object instance variables.
The absence of pointers eliminates memory-browsing, modification of memory resident code, illegal access to security related objects, and over writing the security manager or the class loader. Java is a strongly typed language. Strong typing also contributes to security. Methods cannot be used with classes to which they do not apply. Objects are associated with a well defined types and cannot be freely converted.

The compiler checks all array operations to make sure that they are valid for the array object being accessed and memory overruns do not occur. The compiler checks all class, interface, variable and method accesses to ensure that the accesses are consistent with the access modifiers used in their declarations. The compiler also prevents uninitialized variables from being read and constants from being modified.

## Run Time Security Mechanisms

Of all the security mechanisms of Java security architecture, runtime security mechanisms are the most elaborate and time consuming in terms of affecting the performance. The fundamental assumption here is that all code that is "foreign" to the local system and is 100% untrustworthy, so it should be subjected to extensive security checks even though it affects performance. Performance is secondary to security. As mentioned earlier, this run-time/real-time security functionality coupled with an interpreted execution of the applet degrades the performance considerably. This run-time real-time security mechanism is implemented by 3 VM components:

1 Byte Code Verifier

2 Class Loader

3 Security Manager

The byte code verifier does extensive security checks of the Java byte code to make sure that the byte code downloaded from a foreign server conforms to all VM specifications. In addition to downloading the classes of the applet from the network, the class loader creates and enforces a runtime entity called the "name spaces". The security manager is responsible for authenticating and validating all applet accesses to local resources like the disk, memory, processes etc. The byte code verifier, class loader and the security manager together create a virtual entity called the "sandbox". The sandbox is a virtual "prison" for the applet. It allows an applet to function freely as long as it does not affect any other sandbox. In summary, these three components basically ensure:

- Only the correct classes are loaded
- The classes are in correct format
- Untrusted classes will not execute dangerous instructions
- Untrusted classes are not allowed to access protected systems and resources

The following section describes each of these components in detail. Please see the Java security architecture diagram in Figure 2 for an understanding of the relationship between various security components.

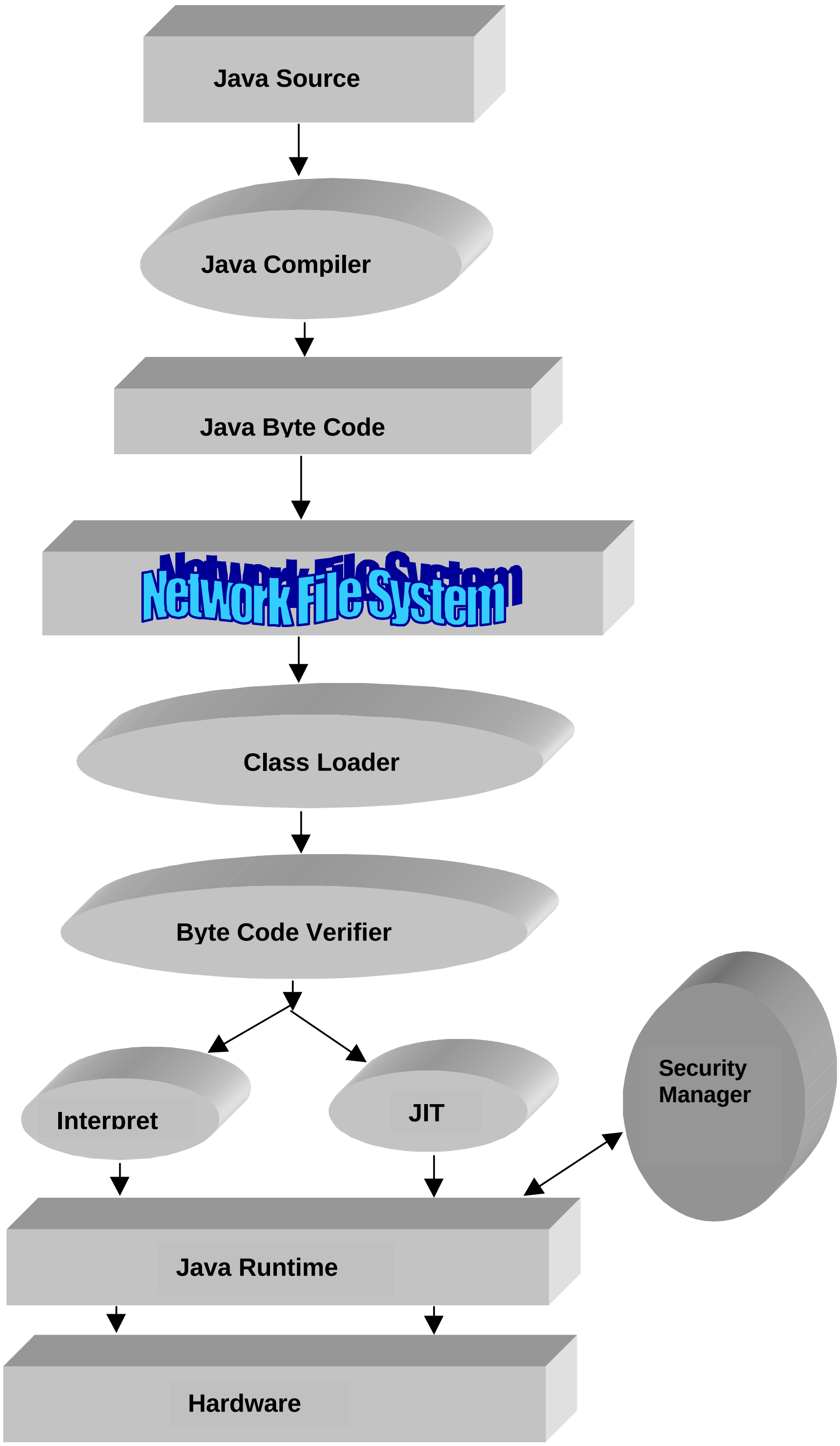


*Figure 2. Java Security Architecture*

## Sandbox

Sandbox comprises of a number of cooperating system components, ranging from security managers that execute as part of the application to security measures designed into the JVM. Sandbox ensures that an untrusted and possibly malicious application cannot gain access to the system resources. In the Sandbox model, a foreign applet is treated as an irresponsible "baby" who is not allowed to move beyond very precise limits: it must strictly respect the semantics of the Java language (byte code verification and access restriction checks), it must not redefine "system classes" with new classes of its own (verified by the class loader of the VM) and it must not try to perform "dangerous" actions such as reading or writing anywhere in the local file system (controlled by the security manager). See Figure 3 for a pictorial representation of the Sandbox model.

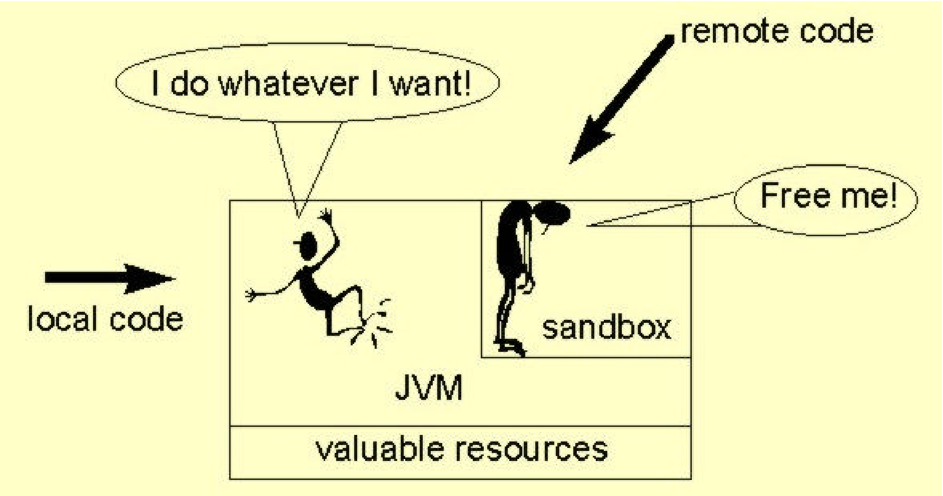


*Figure 3: The Sandbox Model*

## The Class Loader

The class loader prevents classes loaded from the network from masquerading or even inadvertently conflicting with classes that are local. To run an applet, the browser invokes the Java applet class loader, which fetches the applet from the remote machine and creates and places it in a name space created for that applet. The namespace mechanism delineates and controls what other portions of the run time environment the applet can access and modify.

It also ensures that the running applet does not replace runtime components especially the byte code verifier, class loader and the security manager. The mechanism of enforcing access restrictions through name spaces is not new. Operating systems enforce similar access restrictions through address spaces. A process cannot access any location that is outside its address space except through system calls or services. Please note that the name spaces are boundaries of a sandbox.

## Byte Code Verifier

The byte code verifier ensures that the code conforms to virtual machine language specifications and does not violate type and namespace restrictions.

It ensures that the applet does not forge pointers, circumvent access restrictions access objects through illegal cast. In addition, it also checks to make sure that the internal stacks do not overflow or underflow. The verifier uses a mini theorem prover that the .class file initially satisfies certain security constraints and that when executed it will transition into states in which these security constraints are satisfied. It validates all untrusted code before it permits it to execute within a name space. Name spaces ensure that one applet cannot affect the rest of the run time environment. The code verifier ensures that an applet cannot violate its name space.

Please note that the verifier operates in four passes. In pass 1, it ensures that the class file conforms to the class file format. It verifies and validates the magic number and the constant pool. Pass 1 is the simplest of all the four passes. In pass 2, it makes more validation checks on the class file and ensures that the final classes are not subclassed, final methods are not over ridden, checks that all field references and method references in the constant pool must have legal names, legal classes and legal type signature. Pass 3 is the most complex. Byte code of each method is verified. Dataflow and control flow analysis is performed in this phase. Pass 4 does runtime checks. For instance, the first time an instruction that references a class is executed, it loads in the definition of the class if it is not already loaded, verifies that the currently executing class is allowed to reference the given class, etc.

Please note that all these checks and verifications are being done in real-time/run-time. This adds to the run-time of the applet and consequently to the performance degradation.

## Security Manager

The security manager provides a central decision point for Java security rules. It is a class that can be subclassed and a custom security manager can be implemented. In fact, this is true of the class loader and the byte code verifier. A foreign applet is never allowed to replace a security manager or a class loader or a byte code verifier for obvious reasons but a VM implementor is free to implement their own versions of the all three components. This gives the flexibility of tailoring the security model to individual requirements.

The security manager performs run time verification of “dangerous methods”, that is, methods that request file I/O, network access and those that want a class loader. The security manager may exercise veto power over any request. The security manager “polices” the boundaries between the sandboxes, manages all socket operations, guards access to protected resources including files personal data, controls the creation of and all access to OS programs, and processes preventing installation of new class loaders and maintains thread integrity.

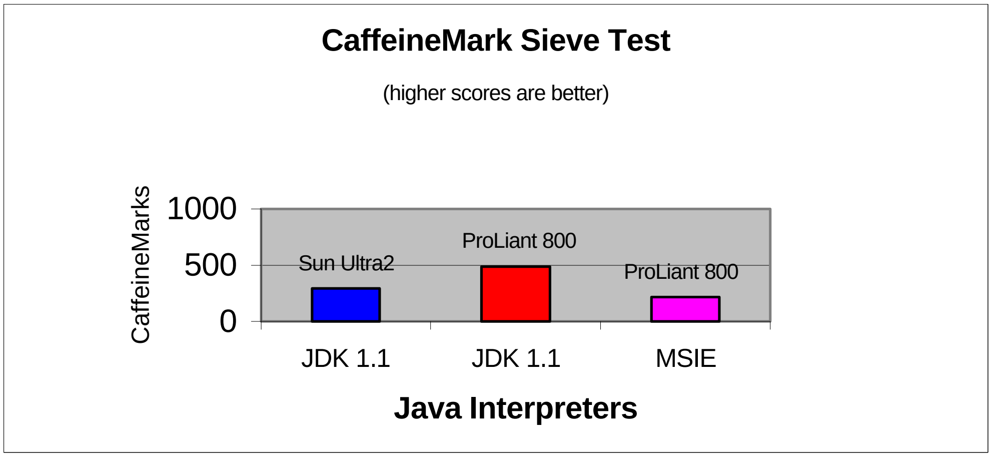
# APPENDIX A

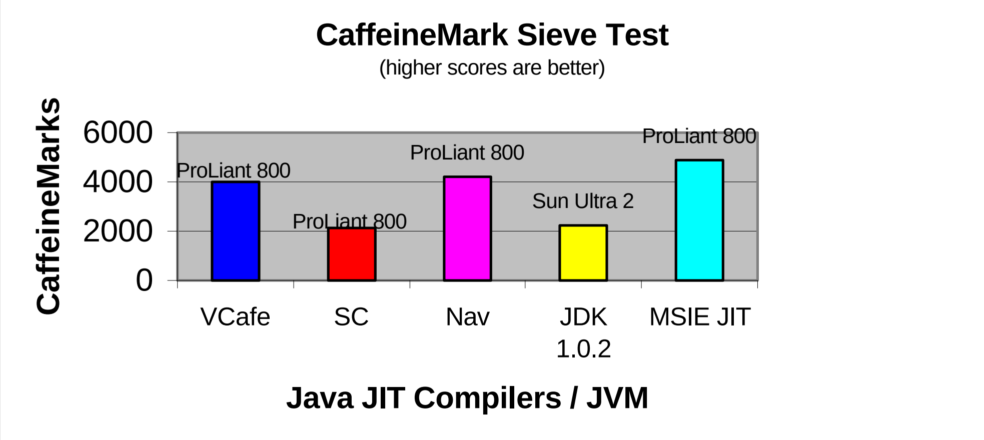
## Details of CaffeineMark Test Results

### Sieve Test

The sieve test is a CPU intensive search for prime numbers below 2048. The test score is proportional to the number of times the search could be performed within the test period.

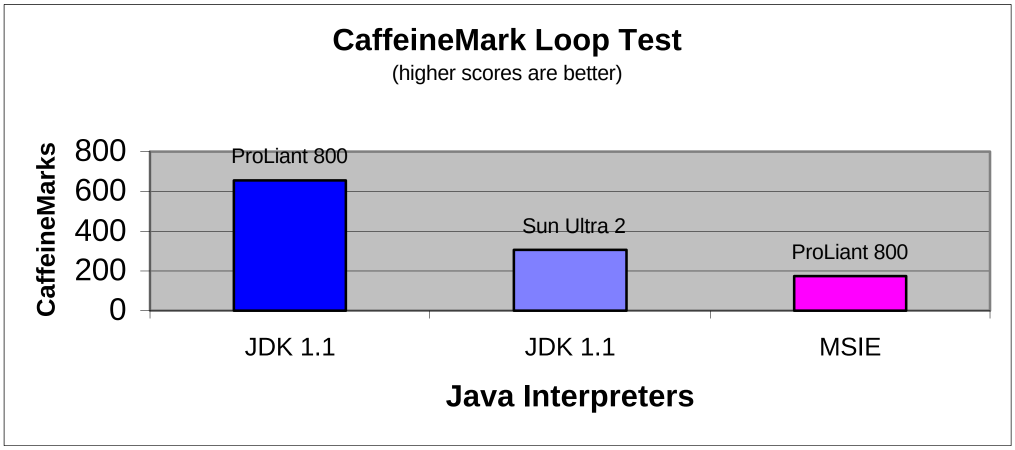
The first bar chart shows the JIT Compiler CaffeineMark scores. MSIE VM JIT Compiler on the ProLiant 800 is 219% faster than Sun's JDK 1.0.2 JIT Compiler. The second bar chart shows the interpreter scores. Sun's JDK 1.1 interpreter on the ProLiant 800 is 165% faster than the Sun's JDK 1.1 interpreter on the Ultra 2 platform. This result shows the performance difference due to the underlying platform (the OS and the HW). The same VM (JDK 1.1) on the ProLiant 800 (NT/INTEL) performs better than on the Sun Ultra 2 (Solaris/SPARC) platform. MSIE JIT on the ProLiant 800 is 22 times faster than the MSIE interpreter on the ProLiant 800 whereas Sun's JDK 1.0.2 JIT is 11 times faster than its own interpreter on the Ultra platform. This confirms the fact that MSIE JIT does far more optimizations of the Java byte code over its interpreter than the Sun's JIT over Sun's interpreter.

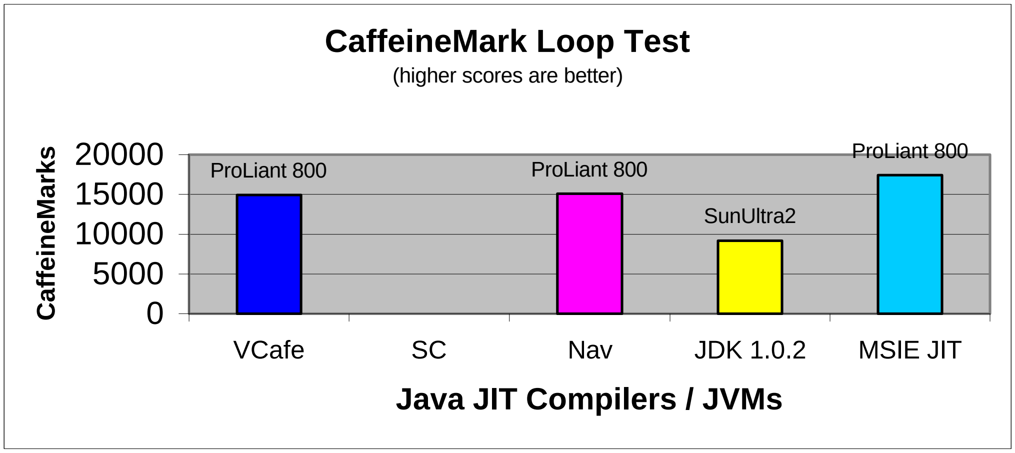




## Loop Test

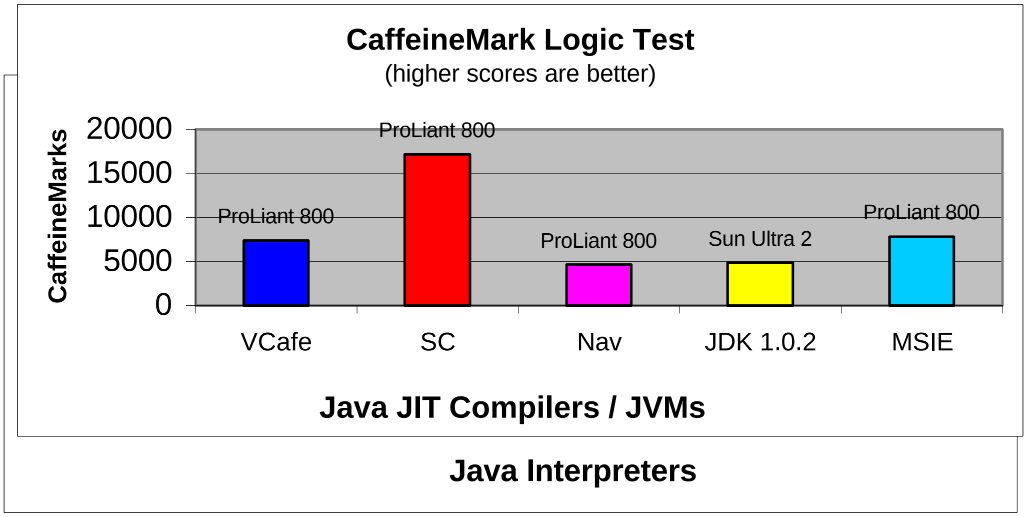
The loop test is sensitive to a number of compiler optimizations. The JIT compilers show their most significant gains on the loop test. Asymetrix's Supecede VM score was omitted from the diagram because its score was extremely high, scoring 359420 on this test, which was found to be an anomaly. MSIE JIT is 190% faster than Sun's JDK 1.0.2 JIT. Sun's JDK 1.1 interpreter on the ProLiant 800 is 216% faster than the same interpreter on Sun's Ultra 2 system. Again comparing the JIT Vs interpreters on the same platform, MSIE JIT is 101 times faster than its interpreter on the ProLiant 800 platform and Sun's JDK 1.0.2 JIT is only 43 times better in performance than its own interpreter on Sun Ultra 2. Netscape' JIT compiler performance is much better and comparable to other JITs on this test than any other test that we ran.





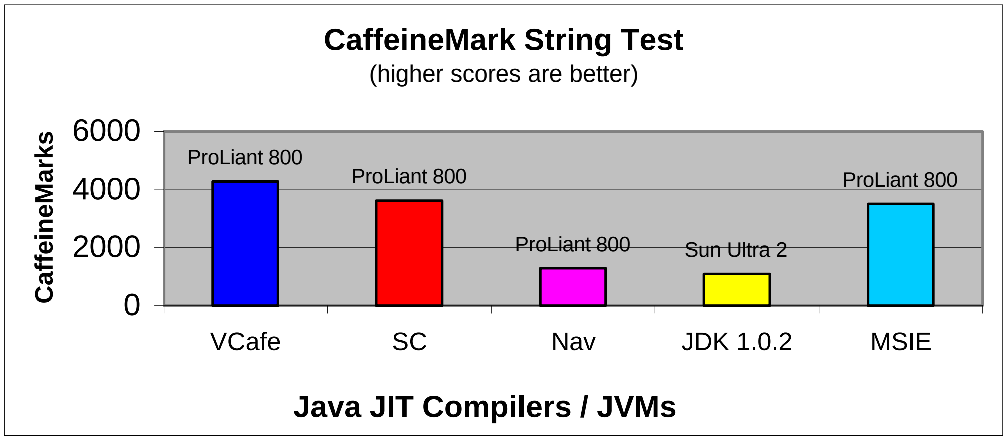
## Logic Test

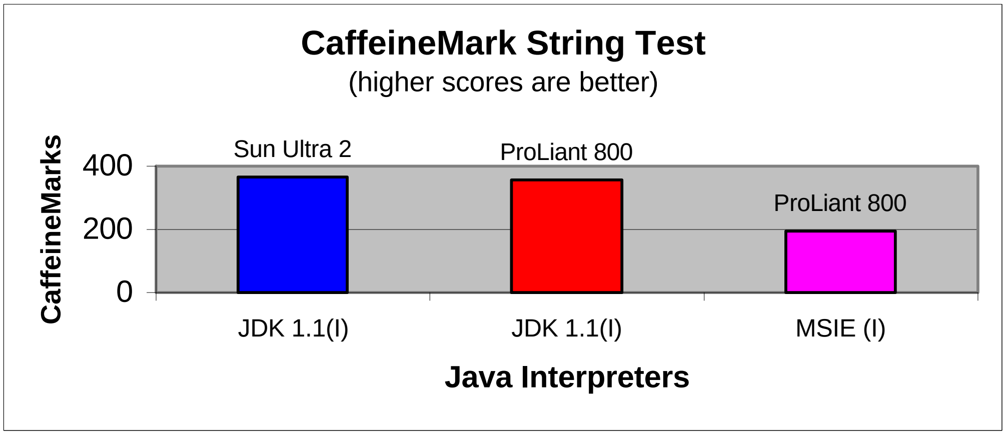
The logic test analyzes the efficiency of the compiler at handling branch instructions. Again the JIT compilers outperform the interpreters. On these tests Asymetrix's SuperCede has performed very well. It has out performed all other VMs or JITs. A comparison of the interpreter performance shows that Sun's JDK 1.1 interpreter on ProLiant 800 is 2.0 times faster than the JDK 1.1 interpreter on Sun Ultra 2. MSIE's JIT compiler's performance is 45 times better than its interpreter and Sun's JDK 1.0.2 JIT compiler's performance is 16 times better than its interpreter on this test.



## String Test

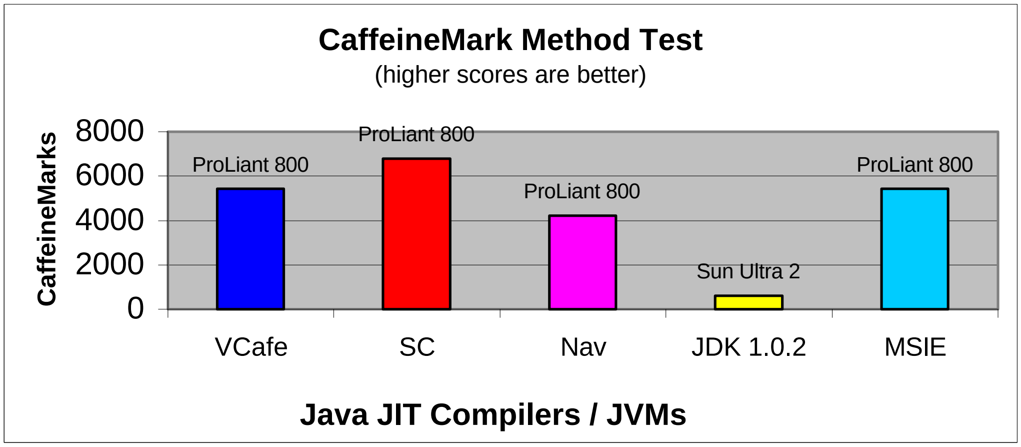
The String Test measures text-processing performance. JIT compilers clearly perform better than the interpreters. Symantec's Visual Café has out scored all other VMs. MSIE JIT is 321% faster than Sun's JDK 1.0.2 JIT and Sun's JDK 1.1 interpreter on Sun Ultra 2 and the ProLiant 800 have the same performance scores. MSIE JIT is 18 times faster than MSIE interpreter on the ProLiant 800 and Sun's JDK 1.0.2 JIT on Ultra 2 is 3 to 4 times better than its interpreter on the same platform.

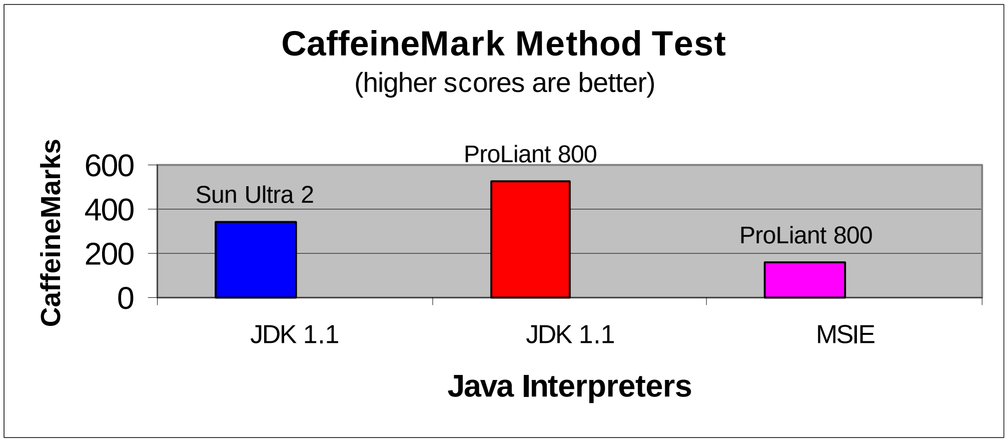




## Method Test

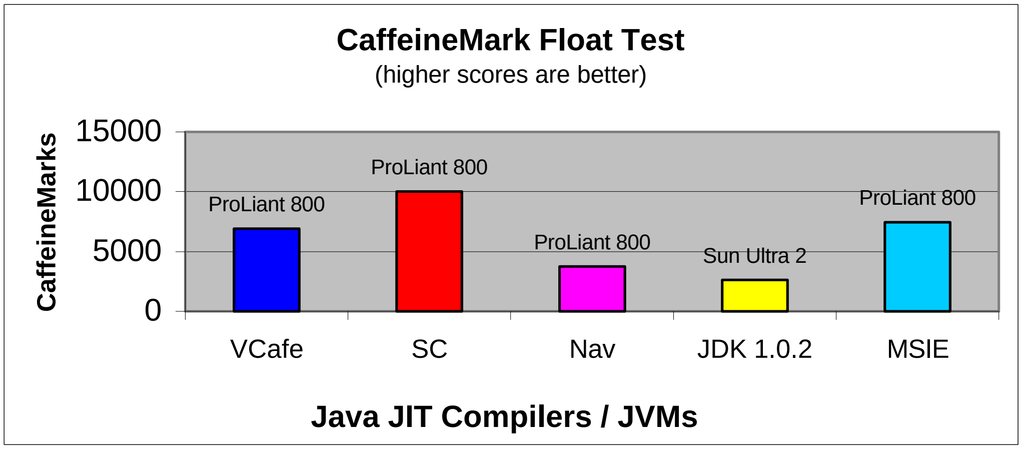
The method test measures the efficiency of method calls in the code. JIT compilers outperform the interpreters. MSIE JIT on ProLiant 800 is 9 times faster than Sun's JDK 1.0.2 JIT on Sun Ultra 2. Explorer JIT on ProLiant 800 is 34 times faster than its interpreter on the same machine. Sun's JDK 1.0.2 JIT is 8 to 10 times faster than the its interpreter on the Ultra system. Sun's JDK 1.1 interpreter on ProLiant 800 is 150% faster than Sun's JDK 1.1 interpreter on the Ultra system.

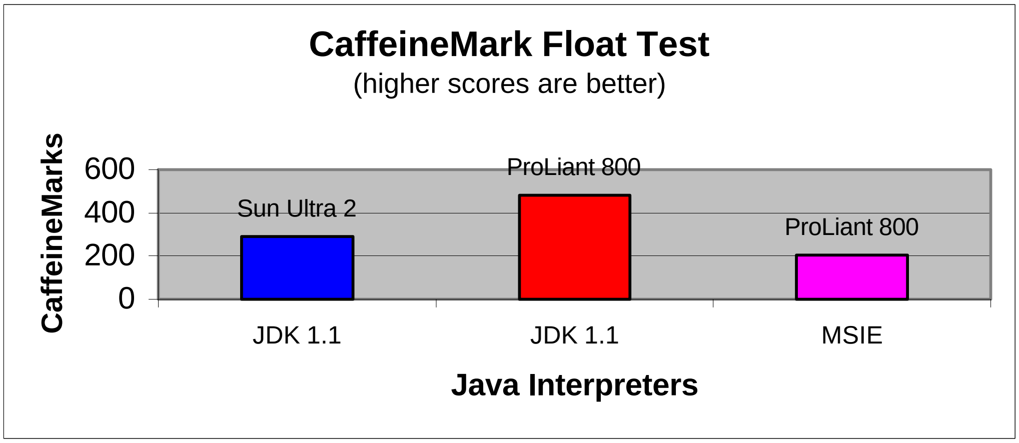




## Float Test

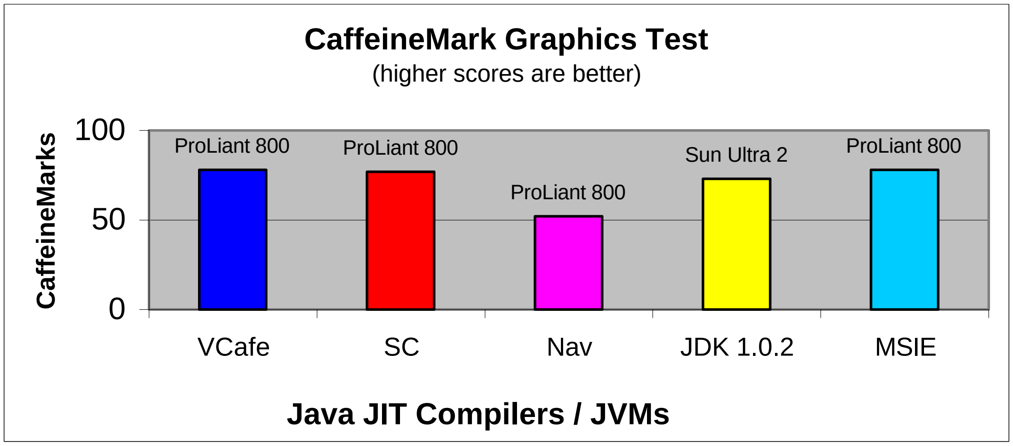
The float test simulates 3 -D rotations of a set of points in space and tests the floating-point capability of the system. The JIT compilers outperform the interpreters as expected**.** MSIE JIT on the ProLiant 800 is 2.8 times or 280% faster than the Sun Ultra 2 JDK 1.0.2 JIT. MSIE JIT is 36 times faster than its interpreter on the ProLiant 800 and Sun's JDK 1.0.2 JIT is 11 times faster than the interpreter. Sun's JDK 1.1 interpreter on the ProLiant  800 is 160% faster than the JDK 1.1 interpreter on the Ultra 2 platform.

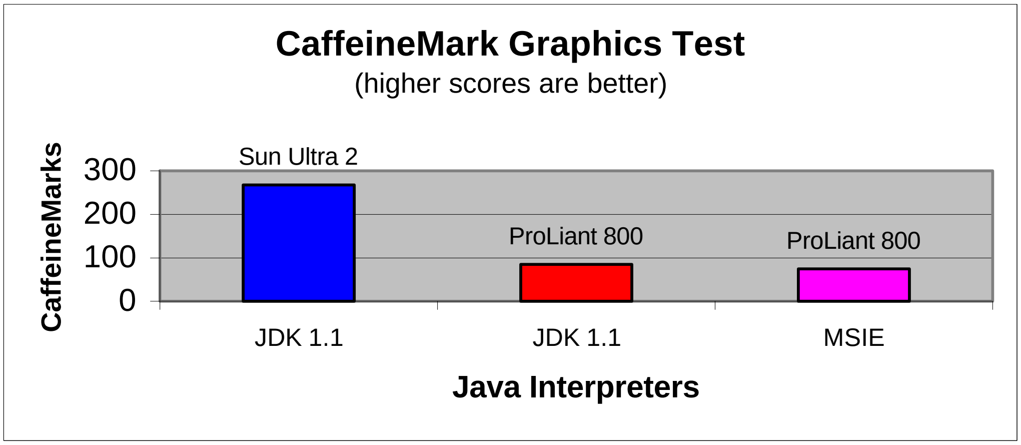


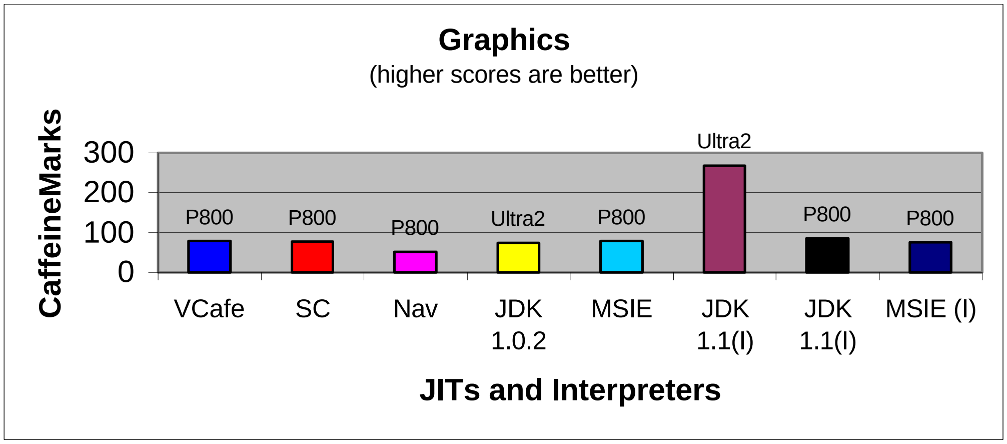


## Graphics Test

The graphics test draws lines and filled rectangles onto the screen using graphics primitives. The graphics, image and the dialog tests, all of which are the tests that exercise the Java AWT package and the graphics subsystem, have produced some very interesting results. The JIT compilers do not produce any significant performance gains over the interpreters unlike the math and compute intensive tests. The MSIE JIT's CaffeineMark score is the same as that of its interpreter. This is also the case with Sun's JDK 1.0.2 JIT and interpreter. The JDK 1.1 interpreter on the other hand is 3 times as fast as JDK 1.0.2 JIT compiler on the ProLiant 800. As mentioned earlier, because of the implementation of most graphics operations in the native code (which also means that the AWT library in Java VMs is an empty library), interpreters have performed better than the JIT compilers as shown in Sun's case on this test. The bar chart on the next page has both the JITs and interpreters on the same chart to provide better visual understanding of the interpreter performance in comparison to JITs on the graphics tests.

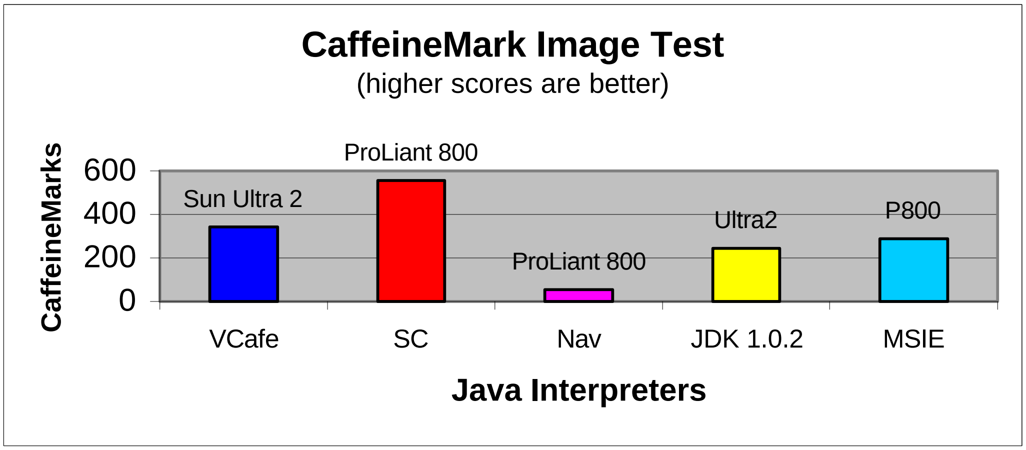


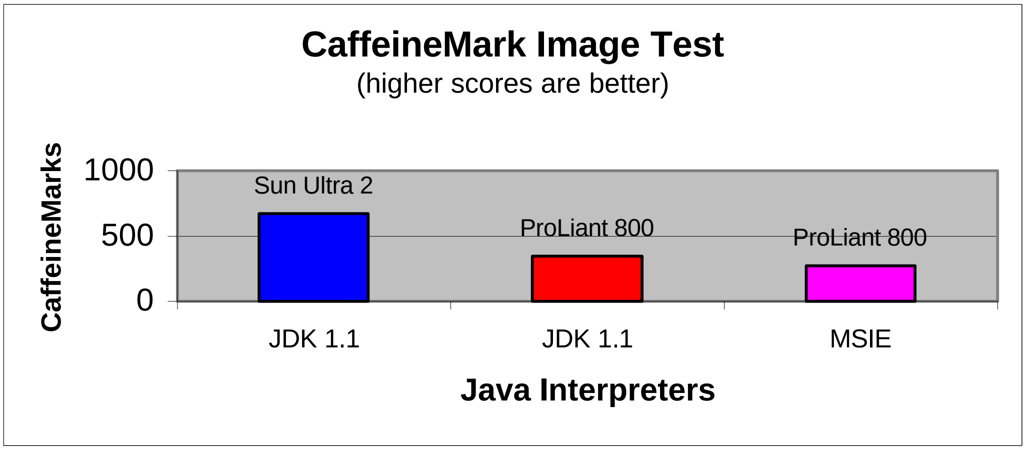


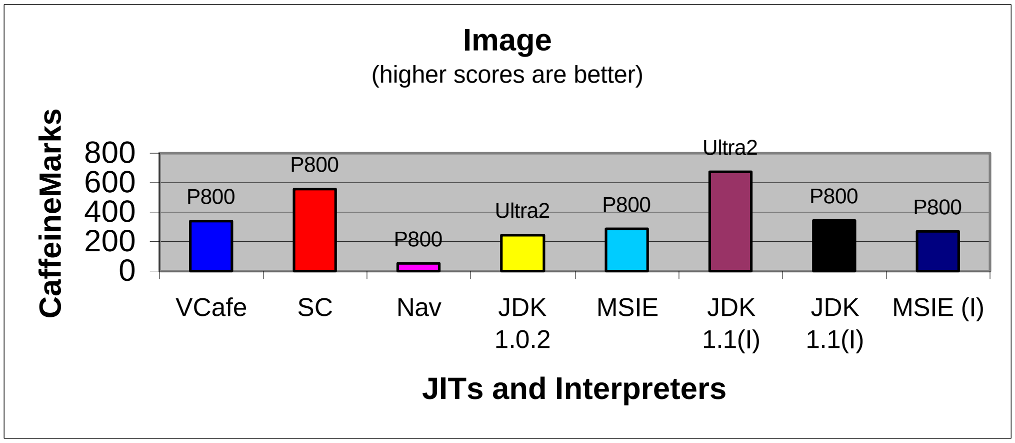

Graphics
(higher scores are better)
CaffeineMarks
300
200
100
0
P800
P800
P800
Ultra2
P800
Ultra2
P800
P800
VCafe
SC
Nav
JDK 1.0.2
MSIE
JDK 1.1(I)
JDK 1.1(I)
MSIE (I)
JITs and Interpreters

## Image Test

The image test draws hundreds of small images into the display area to measure BitBlt speed. The results here are also not very different from the graphics test in the previous section. See the "Image" chart for a JIT and interpreter combined comparative view.

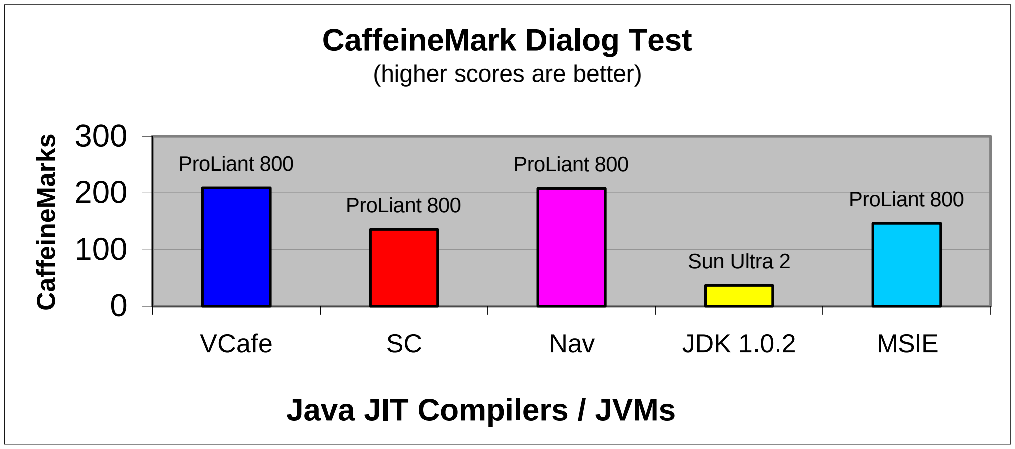


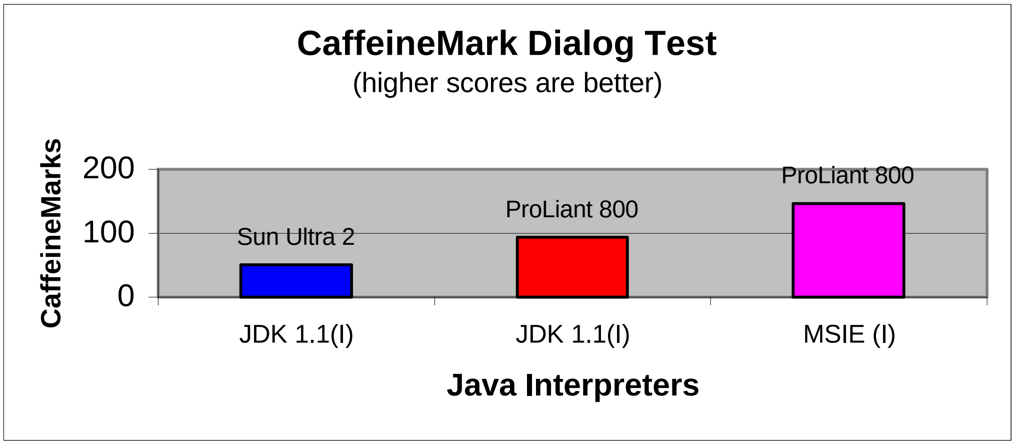


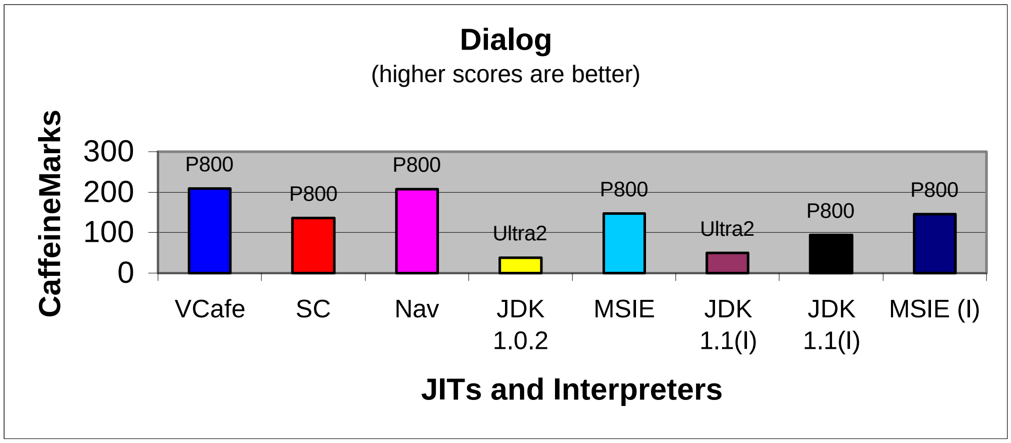


## Dialog Test

To measure label and text box drawing and updating speed, the dialog test creates a small dialog box and updates the labels and test fields. The JITs performed better on this test than the graphics and image tests except the JDK 1.0.2 JIT, which performed the worst. Netscape Navigator and Visual Café did the best. JDK 1.1 interpreter on the ProLiant 800 is twice as fast as the JDK 1.1 interpreter on the Sun Ultra system.

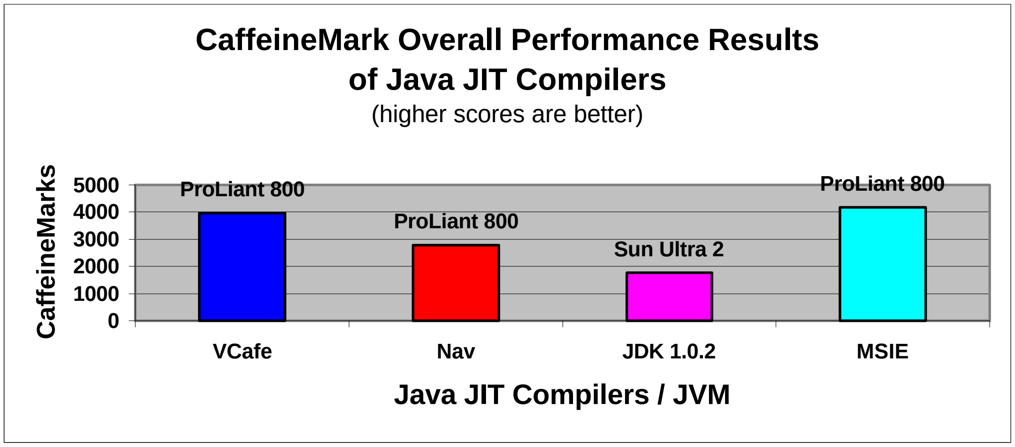


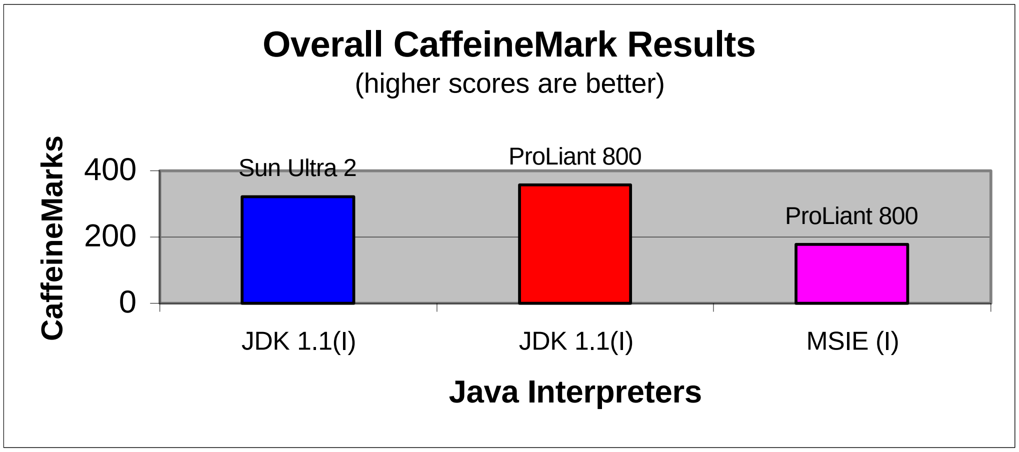


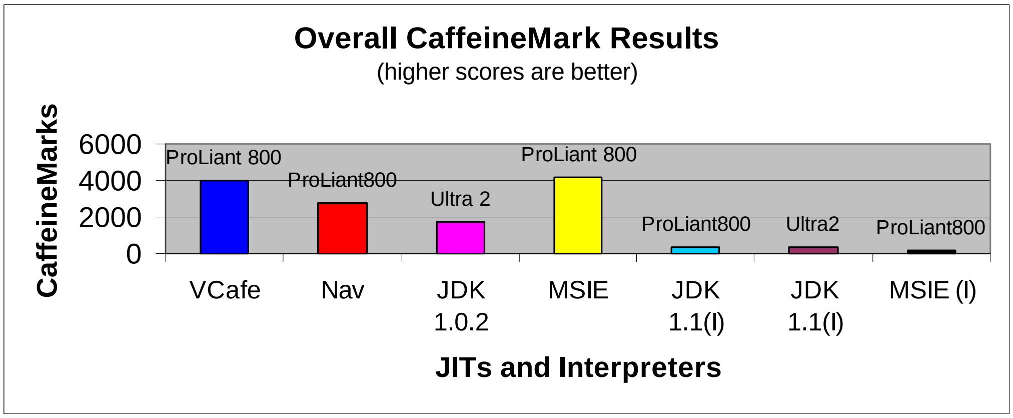


## Overall CaffeineMark

The overall CaffeineMark score shows the dramatic performance advantage of the JIT compilers. The test results of Asymetrix's SuperCede JIT compiler were not included because its overall CaffeineMark scores are skewed due to its loop test results. Please note that the overall CaffeineMark is a weighted average of all the previous tests. It indicates an overall average performance of a VM.

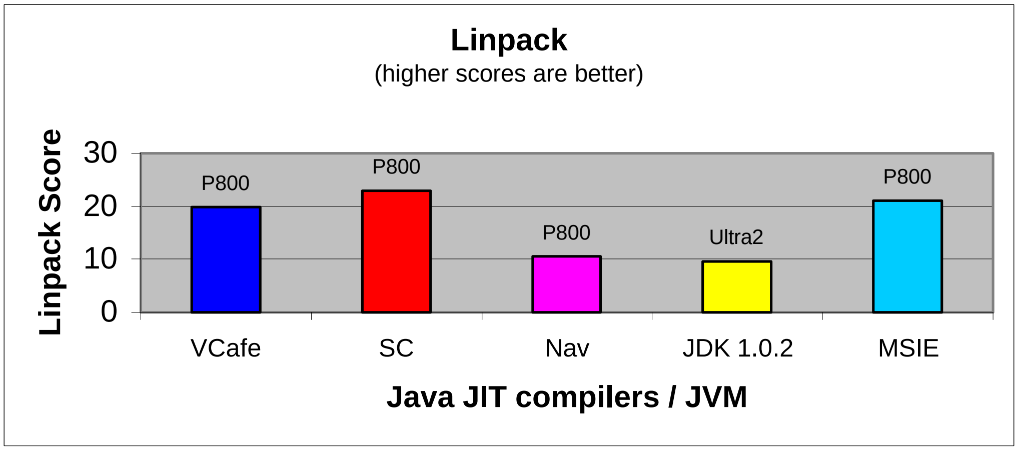


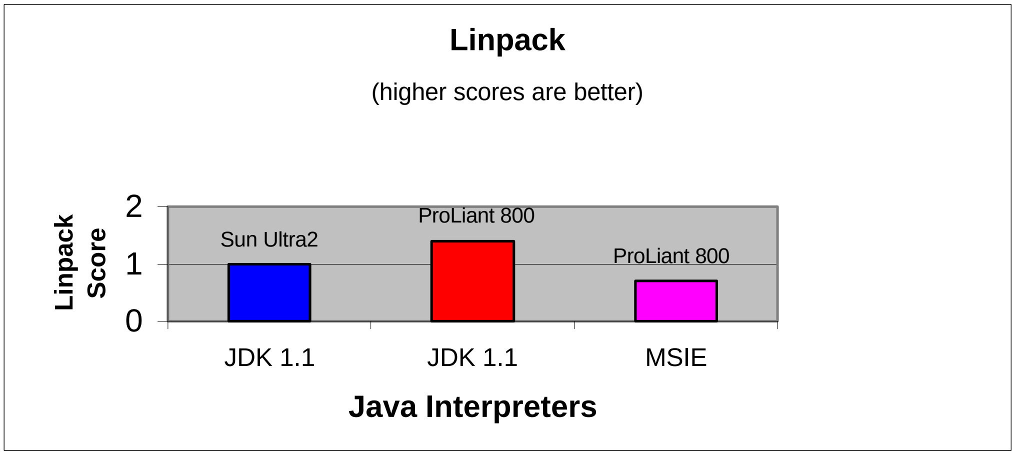


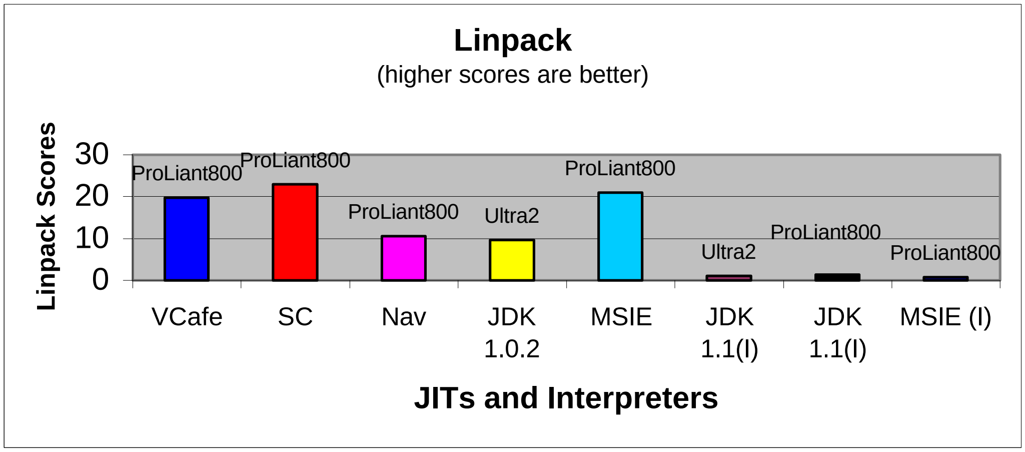


## Details of Linpack Test Results

On this computationally intensive test the JIT compilers speed up performance by a factor of about 20. The three fastest VMs got identical high scores. This may be due to extensive optimization present in the Java bytecode, or it may represent a limitation in the resolution of the benchmark. One of the most interesting effects visible in the Linpack results is the difference between the first and last runs of each VM. While the interpreter scores do not change, some of the JIT compiler scores speed up after the first run. It is believed that the slow down for the SuperCede compiler is just an anomaly of the Linpack benchmark. Linpack scores seem to be quantized, such that, at high performance levels, a small variation in performance can have a disproportionate effect. It is believed that the SupeCede performs the same way each time because it unconditionally compiles all Java code, but the other three JIT compilers "learn" from the initial run and compile Java code as necessary.






Linpack
(higher scores are better)
Linpack Scores
30
20
10
0
ProLiant800
ProLiant800
ProLiant800
Ultra2
ProLiant800
Ultra2
ProLiant800
ProLiant800
VCafe
SC
Nav
JDK 1.0.2
MSIE
JDK 1.1(I)
JDK 1.1(I)
MSIE (I)
JITs and Interpreters

# APPENDIX B

## Details of JMark Test Suite Results

### Introduction

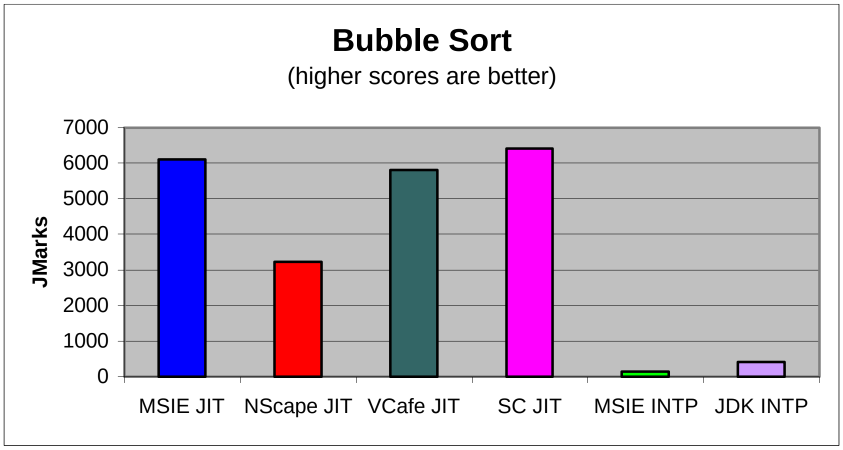
The JMark test suite consists of 11 tests that target specific areas of VM functionality and are implemented in separate Java classes. All the tests, with the exception of the Graphics Threads test, measure the throughput, or the amount of work accomplished in a fixed time. In these tests, a higher score is better. The Graphics Thread test measures the amount of time needed to perform a set amount of work, so a lower score is better. The JMark test result bar charts have both the JIT compiler results and the interpreter results on the same chart unlike the results for CaffeineMark and Linpack test results. This is for easy visual comparison of JIT and interpreter results. Please note that the JMark test suite could not be run on the Sun Ultra 2 system. The tests could be run using the JDK 1.1 VM on the ProLiant 800 system. So the JMark test results compare the performance of various VMs on the ProLiant 800 except Sun's JDK 1.1 where we compared the Sun's VM interpreter with MSIE VM interpreter. This is because Sun's JDK 1.1 release did not have a JIT compiler.

In almost all the JMark tests, Asymetrix SuperCede has performed as well as, if not better than, MSIE and Symantec's Visual Café. As mentioned earlier, SuperCede bypasses the security code in the virtual machine and other VMs that were tested do not so SuperCede results have to be adjusted for this. Based on this, it is felt that the VMs from Microsoft, Symantec Visual Café and Asymetrix SuperCede have similar performance results. Netscape's VM has performed the poorest of all the VMs that were tested. This may be due to Netscape's excessive security checks in the VM.

As expected based on our experience with the CaffeineMark results, JIT compilers showed a substantial improvement in performance over interpreters on the non-graphical tests and the interpreters have performed as well as the JITs, or better in some cases, on the graphics tests.

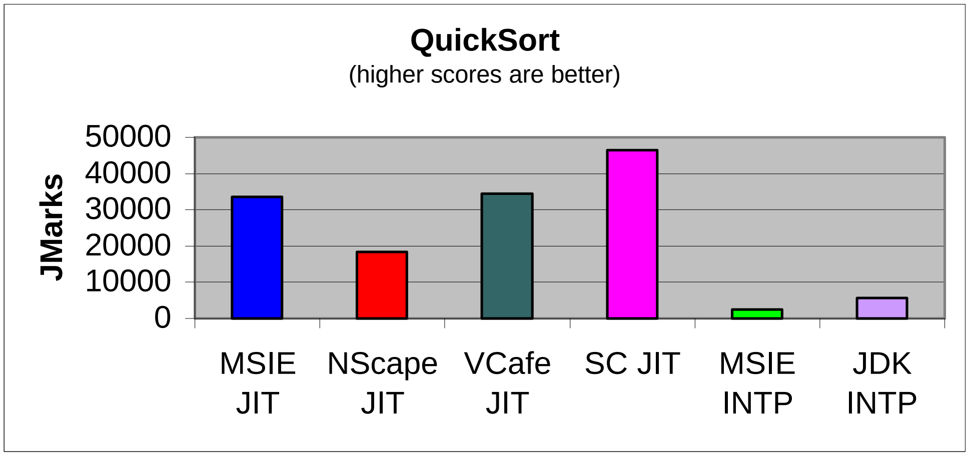
### Bubble Sort Test

The bubble sort algorithm repeatedly swaps out adjacent out-of-order elements in an array of integers until an entire array is sorted. The JMark implementation uses an array of 300 elements. Asymetrix SuperCede VM and Microsoft's MSIE JIT performed the best on this test. MSIE JIT is 40 times better in performance over its interpreter.



## Quick Sort Test

The quick sort algorithm used in this test offers much faster sorting performance on a given array of integers than the bubble sort method. Looking at the graph, it is obvious that MSIE, Visual Café and SuperCede performed the best. MSIE's JIT is 13 times faster than its interpreter.

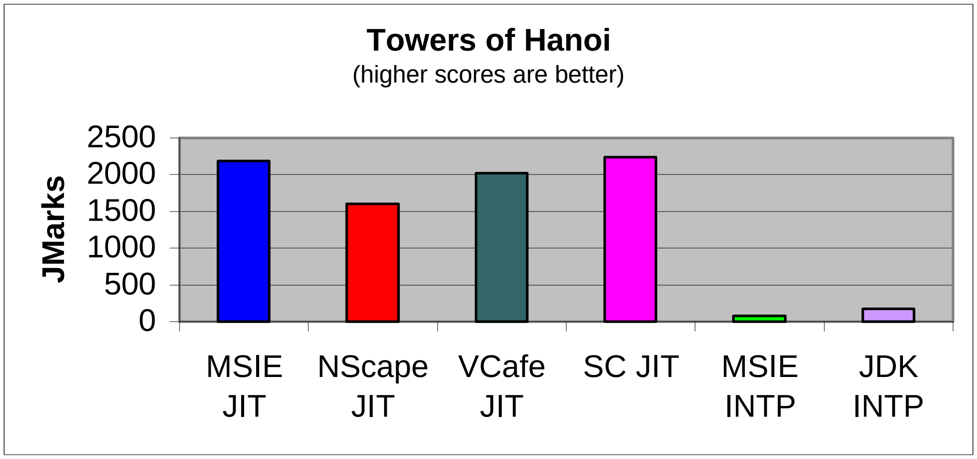


## Towers of Hanoi Test

The Towers of Hanoi test simulates the solution of a classic problem: How to move disks arranged largest to smallest from a first pile, to a second, and then to a third in such a way that only smaller disks are placed on larger ones. The moves are executed one at a time. JMark implementation uses 14 disks and involves over 16,000 moves in an integer array of 3000 elements.

Asymetrix SuperCede and MSIE perform better than all other VMs with Visual Cafe' in a close third place finish.

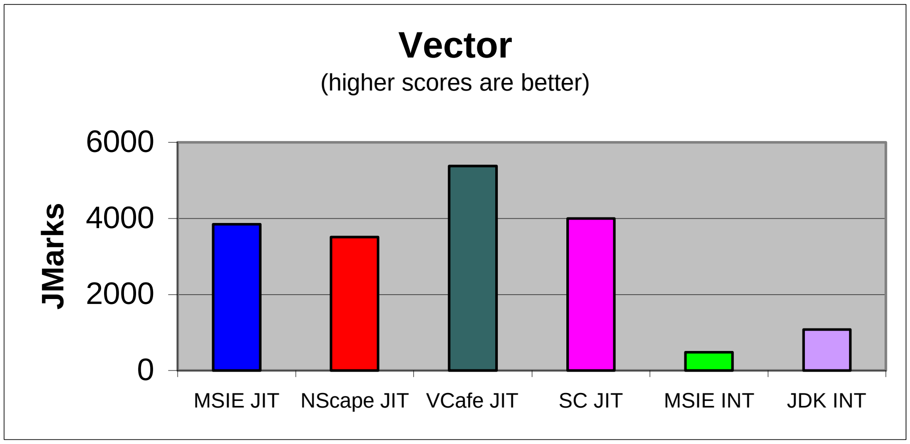
In all the above three non-graphical, computational tests, the performance difference between the interpreter and the JIT speeds is immense. For instance, the MSIE JIT is 0 to 25 times faster than the MSIE interpreter.



## Vector Test

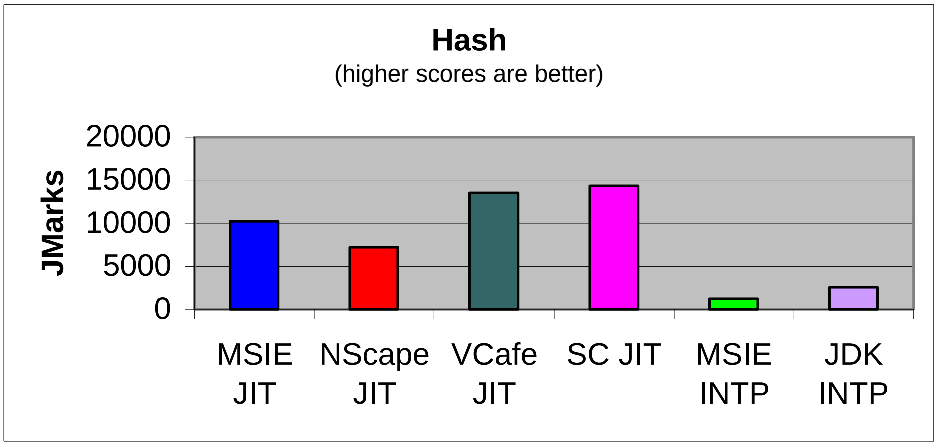
The Vector test exercises the functionality of the Vector container class. This test first adds 100 randomly generated strings of 30 characters each to a Vector object. The test searches for and verifies that each element in the array is present, removes every fourth element, them empties the container. The process is repeated for 100 integer objects using the same sequence of operations.

Symantec's Visual Cafe' outperformed all other VMs with MSIE and Asymetrix in a close second.



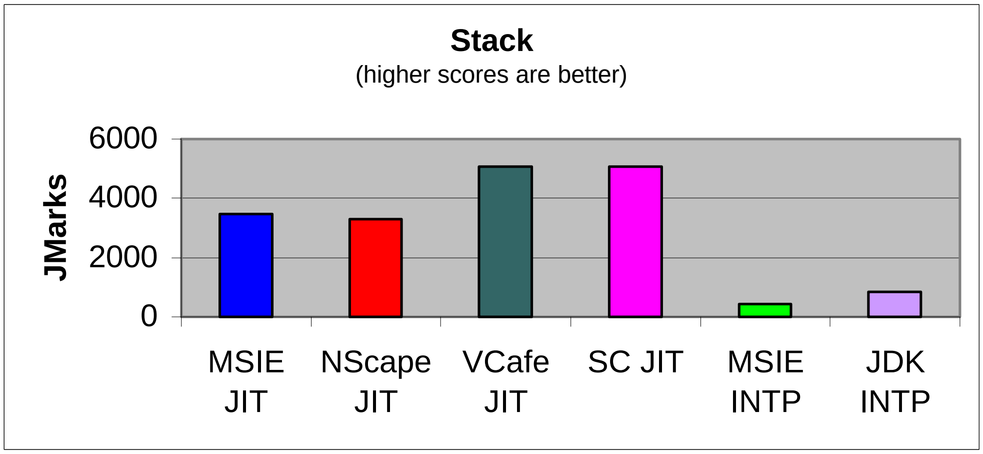
## Hash Table Test

This test makes use of the Java Hash table container utility class. Each element in the container consists of one variable, the key, which is mapped to another variable, the value. The test maps 100 integer keys to 100 String values that consist of 30 characters each. Once the elements are added, every fourth key in the container is searched, then every fourth value. In addition, a quarter of the elements in the Hash table are removed.



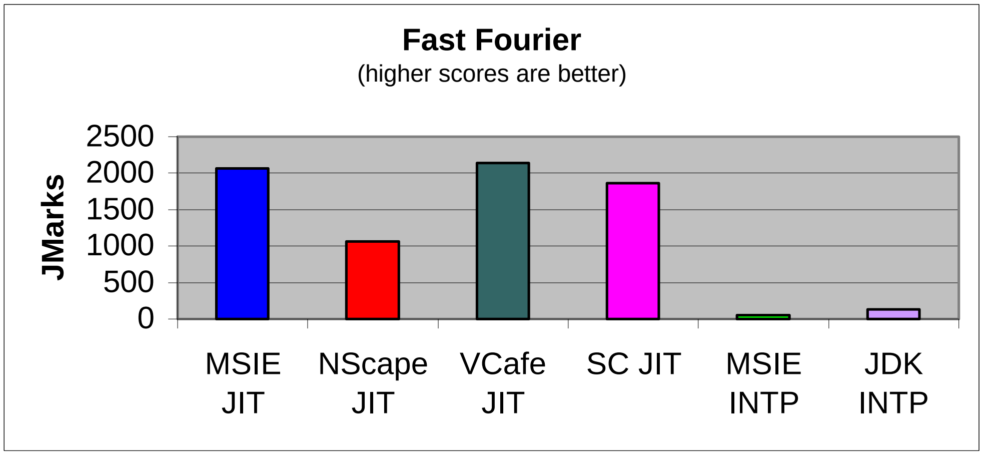
## Stack Test

The stack test is similar to the above two container class tests (Vector and Hash). One hundred strings, each with 30 randomly generated characters, are pushed on to the stack, verify the elements have been added, and then pop these values off the stack. The process is then repeated for 100 integer objects.



## Fast Fourier Transform

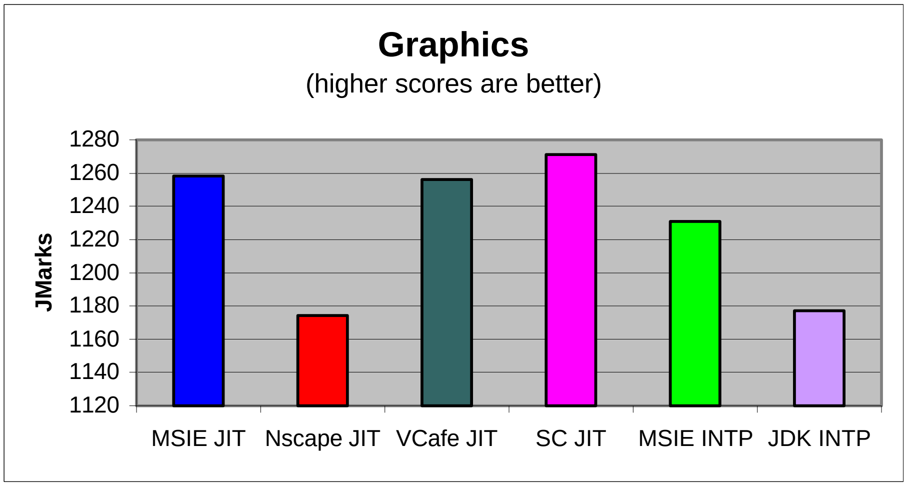
This test simulates a floating-point-intensive matrix operation from the world of physics. This test is an exercise in floating-point calculations and also uses complex arithmetic and heavy array manipulation. MSIE and Symantec's Visual Cafe come out ahead of the pack with Asymetrix's SuperCede in the third place. Again, the JITs outperform the interpreters by a factor of 40.



## AWT Graphics Mix Test

The AWT Graphics Mix test exercises the APIs of the Java Abstract Windowing Toolkit. This test performs a variety of graphics operations sequentially, including drawing lines, rectangles, rounded rectangles, ellipses, polygons and text in different fonts. Where appropriate, these shapes are first drawn using borders only, then as filled-in areas, and finally in 3-D versions with raised edges. These permutations correspond to the capabilities of the Graphics class API methods.

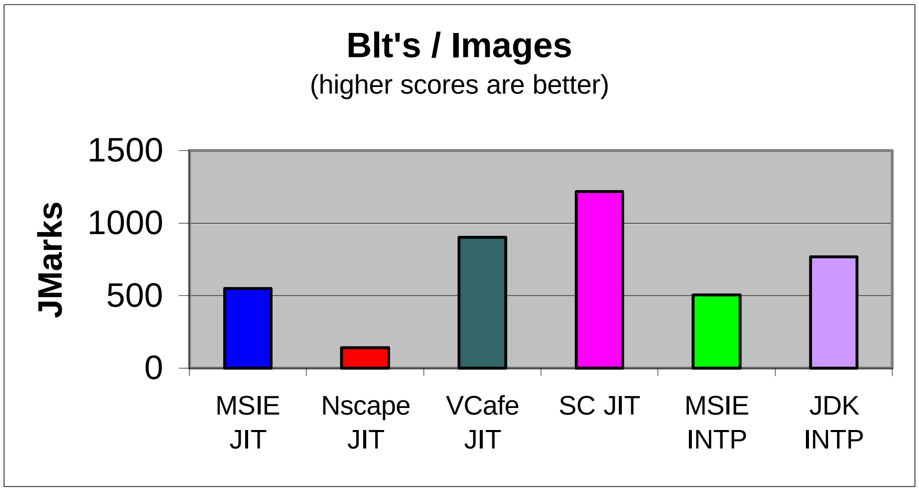
As we have seen with the CaffeineMark test suite, the JITs do not show significantly better performance than the interpreters as in math heavy and data structure access tests. MSIE JIT is 2% better in performance than the MSIE interpreter. The graphics operations are implemented largely in the native code and this could have contributed to such an anomaly.



## AWT Bits and Image Test

This test measures the speed of bit-block transfers, the type of transfers used in animations of graphics data on-screen. This test uses two AWT graphics object methods: drawlmage() and copyArea(). The first call is used to display two GIF images, one 36 by 40 pixels and the other 108 by 120 pixels. These images are first loaded without being timed. They are then bit block transferred to the screen as quickly as possible. In between these image draws, the test makes calls to the AWT Graphics object's copyArea () method, which moves small areas of the screen around, producing a scrambling effect.

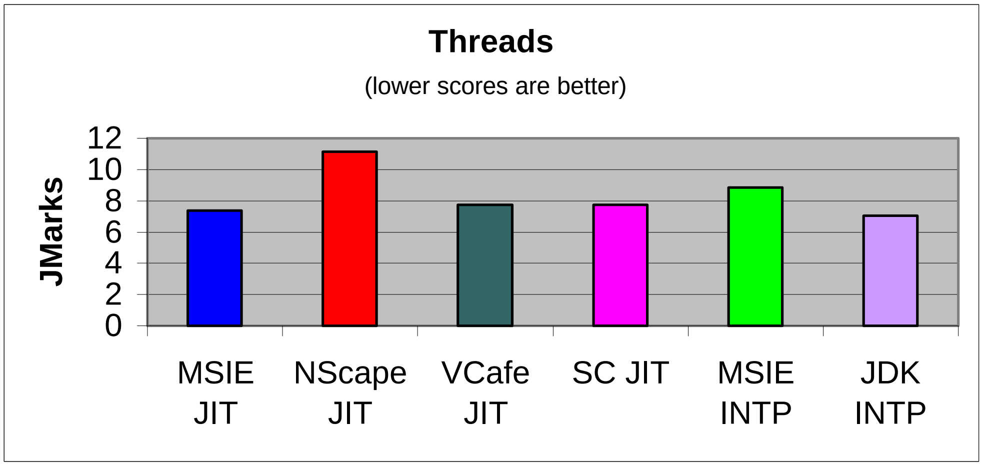
JDK 1.1 interpreter outperforms MSIE JIT and MSIE interpreter. The results on this test are mixed. Some JITs did better than the interpreters (VCafe and SuperCede) and some interpreters performed better than the JITs (MSIE interpreter and JDK 1.1 interpreter over Netscape's JIT).



## Graphics Threads Test

Unlike other tests in the JMark suite, which measure the throughput in a given time, the threads test measures how long it takes to do a fixed amount of work. **So, a lower score indicates better performance**. This test starts 16 worker threads running concurrently. Each one draws semirandom graphics in its own AWT panel on-screen, in a 100 by 100 pixel area.

MSIE JIT outperformed all other JITs. JDK 1.1 interpreter surprisingly outperformed all JITs and interpreters.

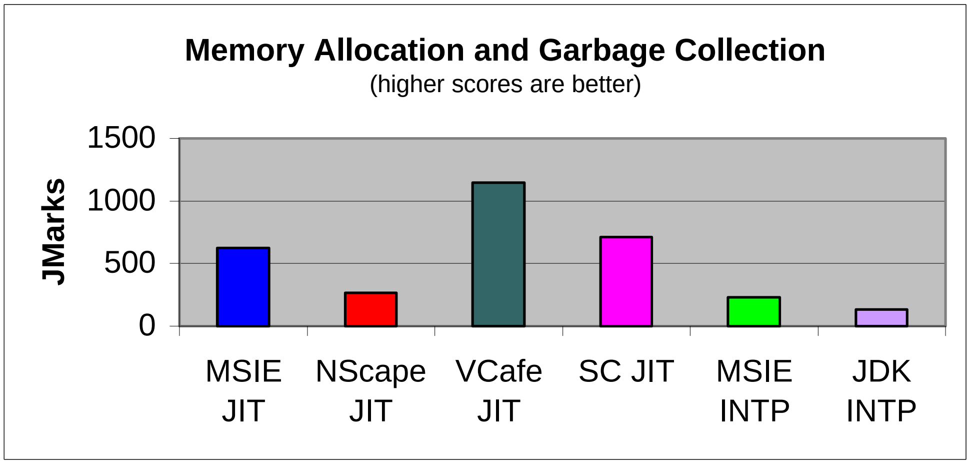


## Memory Allocation and Garbage Collection

The Memory Allocation test measures two characteristics of the Java platforms: the speed of memory allocation and the aggressiveness of garbage collection algorithms.

Symantec's Visual Cafe' outperformed all other JITs by 60% or more in speed.

compilers the intermediate language (IL) is an abstraction of hardware for which the compiler is written. These compilers optimize the IL and generate code for the specific hardware in the backend of the compiler which is then linked with libraries, loaded and executed. Java JIT (Just In Time) Compilers on the other hand optimize the byte code and produce code for the native hardware and execute it at the same time. Please note that the compile, link, load and execute operations are separate and discrete on non-java compilers whereas it is a single continuous dynamic "on the fly" operation in java interpreters or JIT compilers. See Figure 1 for an overview of Java applet processing (i.e. compile, load/download, interpret or JIT compile and run cycle).



## REFERENCES

1 *Secure Computing with Java Now and in the Future*, Java Soft

2 *Security Reference Model for Java Developer Kit 1.0.2*, Marlena Erdos, Bret Hartman, Marianne Mueller

3 *FAQ – Applet Security*, Java Soft

4 *Low Level Security in Java*, Frank Yellin

5 *Java Developer's Guide*, Jamie Jaworski