Build Your Java Apps COMDISCO www.comdisco.com

Performance tests show Java as fast as C++

Java has endured criticism for its laggard performance (relative to C++) since its birth, but the performance gap is closing

Summary
Java got a deservedly bad rap for slow performance when Java virtual machines were bytecode interpreters. With the advent of good just-in-time (JIT) compilers, however, our tests show that Java is already on par with C++ in most areas. And when it isn't, it's for good reason. *(4,000 words)*

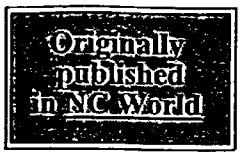
By Carmine Mangione

How does the performance of Java applications compare with similar fully optimized C++ programs in theory, benchmarks, and real-world applications?

Mail this article to a friend

Most industry analysts make the blanket assumption that Java will always suffer performance disadvantages compared with other languages because Java was developed to allow Java programs to run on multiple platforms. You can find this assumption woven through almost all mainstream articles and opinions regarding Java and NCs in modern enterprises.

GOVERNMENT EXHIBIT 1343



So we decided to find out for ourselves just how much of a disadvantage there is between Java and C++, the language to which Java is most often compared. We examined the architectural components of Java and compared the performance of programs written in Java to similar programs in C++.

We expected to see a modicum of lagging performance in Java in each test, though we were skeptical about it running several times slower than C++. To our shock, we rarely found any differences in speed at all. Where Java *is* significantly slower than C++, it's due to Java's stringent security model or to garbage collection.

While we welcome any performance improvements in any language, it appears that if you can use it in the proper context, Java has already come a long way since its inception -- far enough to be considered a top performer along with C++ in many cases.

The test suite

The analysis divides the execution of a program into four functional groups:

- Loading Program Executable
- Running Program Instructions
- Allocating Memory
- Accessing System Resources

These are functions that must be performed by any program running on a computer regardless of its implementation language.

Using these functional groups, we can develop a theoretical performance comparison between programs written in C++ and Java. In addition, we tried different coding approaches to demonstrate the performance characteristics for programs in each group.

The tests are divided into three programs:

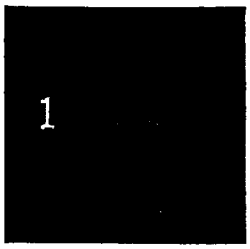
1. The *Simple Loop Test*, which tests performance on function calls, mathematical operations, and up-casting/down-casting
2. The *Memory Allocation Test*, which tests memory allocation and release
3. The *Bouncing Globes Test*, which measures animation performance and handling of resources

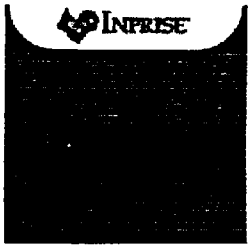
All performance numbers were generated using the following environment:

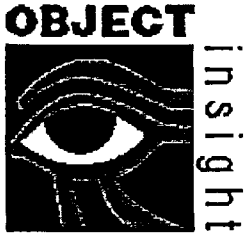
- Platform: Windows NT 4.0 service pack 2
- Hardware configuration: Pentium Pro 200, 128MB RAM
- Software: Visual C++ 5.0 and Sun JDK 1.1.5

For the purposes of this article, we define a "platform" as a combination of CPU type and operating system. For example, Windows NT running on an Intel processor is one platform, Linux running on Intel processor is another, and Linux running on a Digital Alpha processor is still another platform.

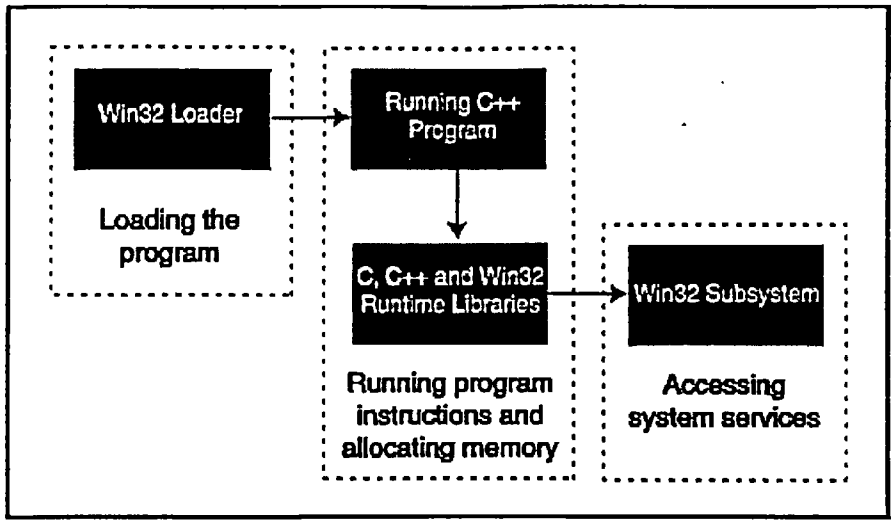
Advertisements



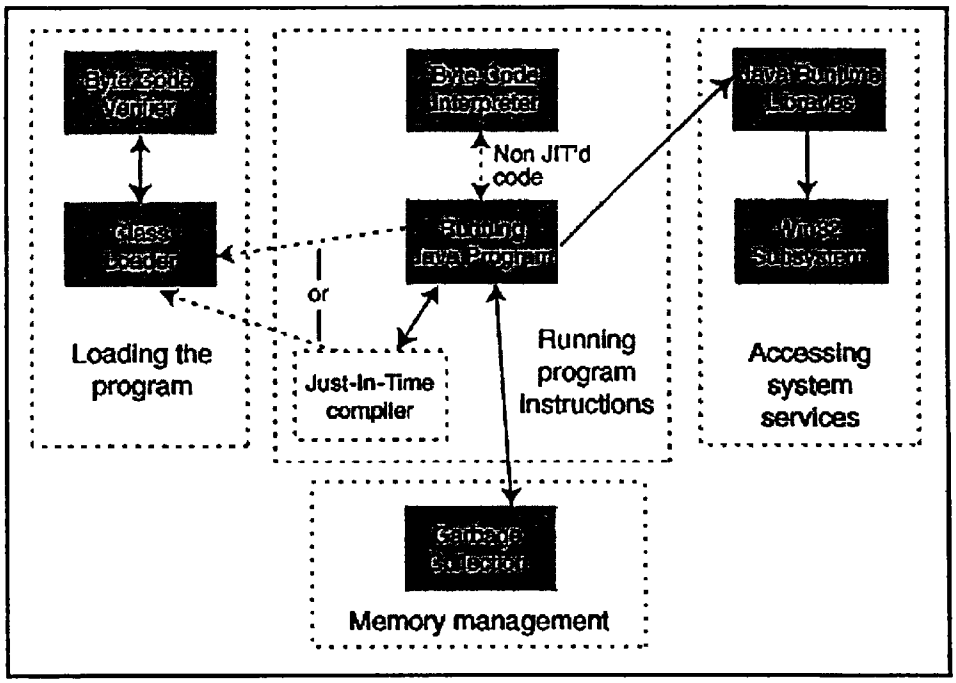




Advertising Information



How the four functional groups in C++ programs are handled on the Windows NT operating system.



How the four functional groups are handled by the Java architecture.

Loading Program Executable (LPE)

Developers create new programs by writing code into one or more source files. A compiler/linker translates these source files into executable files. These executable files can be run on the target machine. The first step in running a program is to load the executable files into memory.

Performance implications: Java versus native C++ (LPE)

If the program is located on a disk drive local to the target platform, loading time of larger programs is rarely of concern. If, however, the program is located on a Web site on the Internet or a corporate intranet, executable size may become the limiting factor in the performance of a program. Over the Internet or intranet, Java programs and resources are, in general, much smaller and faster to load than

native applications. There are two main contributors to this size difference: *executable size* and *selective loading*.

Executable size

Windows NT executables that are written in C++ are significantly larger than similar Java executables. There are three contributing factors that account for this size difference.

First, the binary executable format for C++ programs can inflate code by as much as a factor of two over Java code.

Second, Java provides a series of well-defined consistent libraries, such as mathematics, various network services, collection classes, and graphics classes. The Java virtual machines (JVMs) contain these libraries. In contrast, C++ defines application programming interfaces (APIs) that allow developers to access many of these functions in a consistent manner. Unfortunately, if a developer wants to use any special functions outside the core C++ API, he must deliver the implementation of the supporting libraries with his program. The inclusion of these libraries can double or even treble the size of delivered code.

Finally, Java contains special libraries that support images and sound files in compressed formats, such as Joint Photographic Expert Group (JPEG) and Graphics Interchange Format (GIF) for images, and Adaptive μ-law encoded (AU) for audio. In contrast, the only natively supported formats in Windows NT are uncompressed: bitmap (BMP) for images and wave (WAV) for audio. Compression can reduce the size of images by an order of magnitude and audio files by a factor of three. Additional class libraries are available if you want to add support for other graphics and sound formats.

Selective loading

C++ program loaders must generally load the entire executable file before execution begins.

There are two ways to link DLLs in Win32: statically and dynamically. Statically linked DLLs (by far the most common) are loaded before the program is executed. Dynamically linked DLLs are loaded upon request but are rarely used because the syntax for loading the DLLs and accessing the contained library functions requires a large programming effort in Win32.

In addition, there is no run-time type checking for dynamically linked DLLs. This means that there is no way to tell if a DLL has changed except via a program crash. Finally, if the program is on a remote drive, the DLL must still be brought onto the local machine before it can be used. Establishing an automatic method for doing this in Win32 will require extensive programming on the part of the developer.

In contrast, the Java Loader can selectively load classes in a properly designed program *as they are needed*. For example, consider a full-featured word processor with such features as a thesaurus, a spell checker, mail merge, and export. These features typically produce multi-megabyte files.

The average user will use only a small fraction of the features at any given time. If the program was written in C++, the user would have to load the entire file before proceeding. If the program is written in Java, only those features immediately needed are downloaded, such as the main window. The user downloads additional features only as she needs them.

Real-world example (LPE)

The table below shows the sizes of the programs and resources (if applicable) used in this article. The huge size differences in the C++ versus Java programs are due to the size of the libraries that are required for the C++ program to run. The difference in the resource sizes is due to the compression difference between the GIF files used in the Java program versus the bitmap files used in the C++ program.

Program Name	Program Size: C++ vs Java	Resource Size: C++ vs Java
Simple Loop	46K vs 3.9K	N/A
Memory Allocation	34K vs 1.4K	N/A
Bouncing Globes	103K vs 21K	485K vs 153K

Running Program Instructions (RPI)

Once the program executable is loaded, its instructions must be executed by the target platform's CPU. In traditional C++ programming, these executable files contain binary instructions that are executed by the target platform's CPU. A developer must create a different executable by recompiling the original source code for each target platform. In addition, the peculiarities of each target platform usually forces the developer to modify the original source code.

In contrast, the executable files (called *class files*) produced by a Java compiler contain collections of platform-independent bytecode which cannot be run on a target platform without translation into binary instructions suitable for each target platform's CPU. The JVM is responsible for performing this translation. There are two possible methods a JVM uses to do so: a *bytecode interpreter* or a *just-in-time (JIT) compiler*.

Performance implications: Java versus native C++ (RPI)

Therein lies Java's bad reputation. Most performance perceptions for Java were derived from older JVMs that included bytecode interpretation as the only method for running program instructions.

Bytecode interpreters perform many times slower than comparable C++ programs because each bytecode instruction must be interpreted every time it is executed, which can lead to a great deal of unnecessary overhead. For example, if you code a *repeat loop*, and that loop executes the same set of bytecode instructions many times, the JVM will have to perform the exact same interpretation process on every instruction over and over again, each time it processes an iteration of the loop.

The perceptions earned by early JVMs are no longer valid, since most JVMs are delivered with JIT compilers. A JIT compiler translates and stores the entire class file. This eliminates the need for repeated translations of each bytecode instruction.

Performance among compilers

C++ compilers are able to improve the performance of a piece of code by detecting and improving inefficiencies through a process called *code optimization*. For example, a good compiler can detect if the programmer was sloppy and performed a "static" calculation within a loop. In this case, even though the calculation takes place within the loop, its results remain constant throughout the loop no matter how the program is used.

Recognizing this, the compiler will move the calculation outside of the loop. It will perform this calculation once before the loop is executed, and then use the constant value within the loop without affecting the logic of the program.

This type of optimization is called *expression raising*. The calculation of most optimizations requires knowledge about a group of instructions and may require multiple passes over these instructions. In the expression raising example above, all of the instructions in the loop must be known ahead of time in order to determine if the calculation truly has a constant value for each execution of the loop.

A JVM without a JIT sees each instruction as it is executed, so it cannot perform these types of optimizations on the fly. A JIT can, however, perform code optimization on the entire class file.

As a result, the only significant performance difference between a Java program run with a JIT and a native C++ application will be the amount of time it takes to perform the initial translation of the class file and the types of optimization that are performed.

This overhead will only be a significant proportion of the total execution time if a program is composed of a large number of Java classes that are not used a significant number of times by a program. Real-world programs use the same classes many times, so the proportion of the amount of time spent translating the class will be very low compared to the time spent actually running the code within the class.

Originally, most companies that produced JITs tried to make intelligent decisions about which classes should be compiled and which should not, based on number of times a particular class was used in a program. Many of these companies have since changed their JIT compilers to translate all code, because it turns out the overhead for the translation process is usually insignificant.

Theory and practice

In theory there should be only a negligible difference between JIT-compiled Java bytecode and native C++. In practice, there are two factors that cause performance differences.

First, there will usually be several valid translations of platform-specific instructions when a bytecode instruction is translated into one or more platform-specific instructions. Each of these valid translations will produce the same result, but may have vastly different performance characteristics. If the programmers that create the JIT and C++ compiler are of the same caliber, the performance of both solutions should be similar. (For the purposes of this discussion, we're only considering performance optimizations.)

Second, there is a significant trade-off between compilation time and the number or level of optimizations that are performed on a piece of code. In the expression raising example above, it is usually fairly easy to examine all of the instructions in a loop to determine if a calculation changes throughout the operation of the loop.

In contrast, there is an optimization technique called *dead code elimination* which is much harder to perform. This optimization determines if a particular piece of code is ever used during program execution. If not, it eliminates the code from the executable file.

The performance gains from dead code elimination can be significant, but the overhead of the optimization calculation would most likely be prohibitive in a JIT compiler. It should be noted, however, that dead code is a result of sloppy programming. Competent programmers should be sensitive to the necessity of eliminating any dead code without having to rely upon the compiler to do it for them.

The common optimizations that compilers perform may be divided into groups based on performance gains and computational expense:

> *Primary* and *secondary* optimizations typically afford a program 10 to 15 percent performance gains with minimal computational overhead.
>
> *Tertiary* optimizations can add an additional 5 percent performance gain, but at much greater expense.

This discussion has only listed two of the simpler optimizations that a compiler can perform. For a complete discussion of compilers and optimization theory, see "Compilers. Principles, Techniques, and Tools" and "Crafting a Compiler" (in the Resources section at the end of the article).

Real-world example (RPI)

The table below shows several examples of runtime for a Java routine versus the same functions in C++. Without a JIT, Java performs three or four times worse than C++.

Test	Description	Time (secs) C++	Time (secs) Java (JIT)	Time (secs) Java (Bytecode interpreter)
Integer division	This test loops 10 million times on an integer division	1.8	1.8	4.8
Dead code	This test loops 10 million times and performs an operation that is never used.	3.7	3.7	9.5
Dead code with integer division	This test loops 10 million times and performs an operation that is never used and one that is.	5.4	5.7	20
Floating-point division	This test loops 10 million times on a floating-point division	1.6	1.6	8.7
Static method	This test loops 10 million times calling a static method which contains an Integer division	1.8	1.8	6.0
Member method	This test loops 10 million times calling a member method which contains an Integer division.	1.8	1.8	10
Virtual member method	The Member method test performed above is not really valid. In Java all Member methods are virtual. This test loops 10 million times calling a Virtual member method which contains an Integer division	1.8	1.8	10
Virtual member method with down cast and Run-Time Type Identification (RTTI)	This test loops 10 million times calling a Virtual method on a class that has been down cast using RTTI	11	4.3	12
Virtual member method with *invalid* down cast and Run-Time Type Identification (RTTI)	This test loops 10 million times calling a Virtual method on a class that has been down cast using RTTI	Crash	Crash	Crash

As predicted by the theoretical evaluation, Java with a JIT performs precisely like C++. The only exception is the performance of the Run-time type identification (RTTI) under C++.

Complex object-oriented programs contain complex hierarchies. These hierarchies often require that one program cast for a parent class to a child class. This type of cast is called a *down cast*.

Run-time type identification identifies when a parent class is cast into an invalid child class. Without RTTI, complex systems often experience subtle bugs that are difficult to detect and can compromise the entire project.

Most C++ programmers turn off RTTI due to its expense. Java does not allow programmers to turn off RTTI since this would violate the Java security model. In these tests, it is evident that in C++, RTTI is very expensive as compared to Java.

Allocating Memory (AM)

Programs must allocate memory to store information and perform calculations. When the program is through using the memory, it must release it back to the system so that it will be available to other parts of the system.

C++ and Java allocate memory in much the same manner. C++ programs must explicitly release memory back to the system. One major cause of bugs in C++ programs is that programmers forget to explicitly release memory back to the system. This memory is "leaked" and will not be available until the program terminates.

In Java environments, there is a subsystem called a *garbage collector* that detects when a program no longer needs a piece of memory, and consequently releases it back to the system.

Performance implications: Java versus native C++ (AM)

Since the garbage collector in Java has to be able to determine which pieces of memory are no longer in use, the overhead of memory management for simple tasks is much greater in Java than C++. However, there are two advantages to the Java garbage-collection model that C++ lacks.

First, your programs are virtually immune to memory leaks. Since memory leaks are a frequently occurring bug in large-scale systems, this greatly reduces development time, since you don't have to spend lengthy debugging sessions finding and squashing such leaks.

Second, memory fragmentation can be a major problem in large-scale systems. Memory fragmentation occurs when large numbers of memory allocations are made and released and can seriously hinder the long-term performance of an application. A well-written Java garbage collector can move allocated memory around and prevent fragmentation.

So, although there is an acknowledged trade-off between Java and C++ regarding memory allocation, the trade-off results in great benefits for Java at the price of the performance lost.

One problem with a garbage-collection strategy is that the garbage collector must be able to locate unused objects in memory. If the program tightly adheres to the object-oriented philosophy of containment, the process of garbage collection will be relatively cheap. If, however, the programmer users a procedural approach, it will be difficult for the garbage collector to figure out which objects are not in use due to the complexity of uses relationships.

Real-world example (AM)

Allocating and freeing 10 million 32-bit integers took 0.812 seconds in C++ and 1.592 seconds in Java. This example detects the overhead of the Java allocation system. Even with all of the extra work that the garbage collector has to do, Java's performance in this area is reasonable when compared with C++.

Accessing System Resources (ASR)

In addition to allocating memory, a program requires access to system services such as drawing, sound playing, and window-management routines. Traditional C++ programs access these services through the use of system services APIs. These APIs are unique to each platform forcing the developer to make extensive modifications when porting applications.

Java programs access system services through a single API on all platforms. Each platform contains peer interfaces that map the Java calls to the ones that are available on the platform.

Performance implications: Java versus native C++ (ASR)

In general, the overhead of a call to the system service vastly outweighs the cost of a peer interface in

Java. Java programs that use system services perform as well as native C++ programs.

The exceptions to this rule are those Java APIs with no native equivalent on the target platform. For example, since the Macintosh does not have any native thread support, it is quite painful to use the Java thread API on a Macintosh.

Real-world example (ASR)

This example is an animation engine created by animation and multimedia guru Jay Bartot. The performance of the Java version in Sun's applet viewer is equal to that of the native Win32 program written using the Win32 SDK.

This example does raise two important points.

First, Java performance is only as good as the JVM. On Windows NT, this program does not perform well under Internet Explorer 4.0 or Netscape Communicator. The animations are stilted and jumpy. When run in these browsers under Windows 95, however, the performance comes close that of the native application.

The performance difference of these browsers represents the browser developers' desire to optimize performance on Windows 95 rather than the embattled Windows NT. Due to the differences in the architectures, it's difficult, if not impossible, to make a complex program like a browser perform well on both operating systems.

Second, Sun's JDK for Win32 is a reference standard that other browsers and JVMs should strive to meet.

Competent preparation prevents poor performance

There seems to be no end to the number of industry spokespersons who will deride Java for poor performance. The same phenomenon occurred when analysts derided the ascension of second-generation languages, then structured programming, and now object-oriented programming. Although each new innovation allows larger, more robust programs, each technology demands a new discipline. Since Java is an object-oriented language, most resistance to its acceptance comes from those who don't understand the advantages of a new paradigm.

This analysis has shown that, in theory and practice, there is rarely any significant performance difference between native C++ and Java applications. And when there is a difference, the Java programmer reaps benefits the C++ programmer does not.

Java provides three such advantages over native C++ programming that will greatly reduce the development time of and enhance the performance of large-scale applications:

- The reduction of executable and resource size
- The availability of consistent, robust libraries on most platforms
- The use of garbage collection to eliminate resource leaks

Also this month in JavaWorld Go

Resources

- "Compilers: Principles, Techniques, and Tools" by Alfred Aho, Ravi Sethi, and Jeffrey Ullman,

1986, Addison-Wesley Publishing Co.
http://www.clbooks.com/sqlnut/SP/search/gtsumt?source=javaworld&isbn=0201100886

- "Crafting a Compiler," Fischer, Charles N., LeBlanc, Richard J. Jr., Benjamin Cummings Press, 1988
http://www.clbooks.com/sqlnut/SP/search/gtsumt?source=javaworld&isbn=0805332014
- Give us your opinion on the difference in performance between Java and C++. Take our latest reader poll.
http://nigeria.wpi.com/cgi-bin/gwpoll/gwpoll/ballot.html

About the author

Carmine Mangione is a senior software developer for Netbot Incorporated in Seattle, WA. Reach Carmine at carmine.mangione@javaworld.com.

©1996-98 Web Publishing Inc./IDG—Click for Trademarks & Notices

If you have problems with this magazine, contact webmaster@javaworld.com
URL: http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html
Last modified: Wednesday, September 02, 1998