High-Performance Java™ Software Development

James C. Schatzman and H. Roy Donehower
TRW Systems

Java is a computer programming language designed for ease of programming, object orientation, elegance, and reliability. Written as an interpreted language, it is not designed to be highly efficient. It can be very slow, compared with C/C++ or FORTRAN. For computationally demanding algorithms, simple implementations in Java can run at least 100 times slower than C++. Does that eliminate Java as a serious option for software projects where execution efficiency is an issue? We have found that, with care, most software can be written to perform within a factor of 2 of C/C++. Some kinds of algorithms remain difficult to code efficiently in Java, but they can generally be isolated and coded in another language, then accessed via a Java Native Interface. Here, we examine Java performance in several key areas and suggest ways to enhance it.

Introduction

Most new software products must be object oriented (OO); use component architectures; be portable; entail low development, deployment, and maintenance costs; promote effective software reuse; exhibit high reliability; and support Web enablement and extensibility. Such requirements—particularly those related to object orientation and those requiring excellent portability across UNIX/Windows® platforms and Web enablement—weight the choice of computer language heavily toward Java and away from C/C++ or others. However, Java development can carry a performance penalty. We evaluate several effective strategies to achieve very high performance for many types of applications in Java software. Indeed, some applications, including data visualization and virtual reality, can perform much better than legacy systems based on UNIX and X-Windows.

Are there compelling reasons for software development in Java, instead of the time-honored C and C++? After all, object orientation and Web enablement can certainly be accomplished in C++. Unlike C++, however, Java was designed as an OO computer language from the ground up. It embodies, encourages, and supports OO development in ways that are impossible for C++. Further:

- The nearly 100% portability of Java object code (Java byte code) is observed across UNIX and Windows platforms. Such portability has never been possible with C++ object code and is extremely difficult to achieve for large software systems using C++ source code.
- Through its applet mechanism, Java is designed for use in stand-alone applications, as well as specifically for Web programming.
- Java supports component architectures through its simpler-than-C++ interfaces, simple dynamic linkages, and extremely easy-to-use interprocess communications.

With minor attention to detail, developers can easily write software that works in both stand-alone and client-server modes.

Recent data suggest Java development can be substantially cheaper than similar C++ projects. In a 1998 study of nine companies that developed Java-based systems, International Data Corporation found Java returned a 25% cost savings, compared with C++ [1]. Cost savings during the coding phase averaged 40%. Two features of Java were instrumental: automatic memory management and simplified object orientation. Those findings support our observation that software development can be significantly accelerated with Java's automatic memory management and garbage collection, exception-handling features, and free application programming interfaces (APIs) such as Java 3D™, Swing, and JGL. Our own experience indicates two to four times higher individual developer productivity.

Another advantage of Java comes from low deployment costs. On a per-CPU cycle basis, WinTel platforms are three to ten times cheaper than popular UNIX (Sun Microsystems, Silicon Graphics, Inc.) workstations. Wintel platforms support vast quantities of popular and inexpensive commercial software. Users are generally comfortable with the platform. Those strengths and others make Java extremely attractive for new projects. Government and commercial customers are also beginning to prefer Java solutions.

However, Java has a well-deserved reputation as being slow. Indeed, we present cases where Java runs at least 100 times slower than corresponding C++. Some believe Java technology will eventually reduce the Java performance penalty to insignificance. We have found that some software runs nearly as fast as C++ but is hindered by severe performance problems with certain algorithms or operations. We demonstrate that, if reasonable guidelines are followed, many software projects can be implemented effectively in Java and achieve high performance.

Java Performance: Pure Computation

Software developers debate whether Java is slow, and by exactly how much, compared with C/C++, FORTRAN, or other languages. Java is partly interpreted by the Java Virtual Machine (JVM) and partly compiled into machine code by one or another just-in-time compiler (JIT). Early implementations of Java were indeed slow, but current JITs often deliver excellent performance. Table 1 shows perhaps surprising results for two computationally intensive algorithms.

Table 1. Run Times for Two Computationally Intensive Algorithms

Deployment	1024-Point FFT: Variant A (ms)	1024-Point FFT: Variant B (ms)	1-Million-Point Quick Sort: Variant B (s)
FORTRAN	N/A	2.020	0.820
C++	6.26	0.611	0.800
Java			
Client Hotspot JIT	4.95	0.827	0.897
Server Hotspot JIT	4.59	0.732	0.921
No JIT	163.00	8.910	5.830

The hardware platform for these benchmarks is a 1-GHz Intel Pentium III Coppermine with a large main memory (no paging). Java Version 1.3.1, Microsoft® Visual C++ Version 6, and GNU g77 Version 0.5.19 were used in all cases. We implemented the algorithms in the three languages as similarly as possible. Variant A of the fast Fourier transform (FFT) implementation created new *Complex* objects dynamically. Variant B reused *Complex* objects with an in-place algorithm. In practice, a JIT is always likely to be used; in such cases, slowdowns for Java implementations versus C++ ranged from about 10% to 35%. Variant A ran faster in Java than in C++, suggesting Java was faster than C++ at dynamically allocating large numbers of small objects—interesting, but the practical conclusion is that variant B is to be preferred in either language. Why may Java execution be slow, even when it is compiled to machine code by a JIT? Potential explanations include the following:

- *Overhead.* Java objects carry certain overhead data, such as thread synchronization information, that C/C++ objects or data structures avoid. A purely memory-organizing data structure (C struct) does not exist in Java. For example, complex numbers represented by Java objects are twice the size of FORTRAN complex numbers, and execution overhead is involved as well.
- *Referencing.* Java arrays of objects are actually arrays of *references* to objects. This additional level of indirection must be traversed by Java array-handling code. Additionally, memory used by a Java array of objects—such as arrays of complex numbers represented by arrays of *Complex* objects—may be highly nonlocalized and therefore less efficiently operated on by modern memory systems.
- *Memory allocation.* Straightforward OO—especially Java—implementations of complex-number arithmetic packages tend to involve much dynamic creation of new objects. For example, a complex product would normally produce a new *Complex* object, allocated on the heap. This implementation is less efficient than using statically allocated memory, stack memory, or processor registers for storage of intermediary results.
- *Range checking.* Java array operations unavoidably incorporate subscript range checking. It is not possible to turn off range checking in current JVMs.
- *Portability versus optimization.* JITs are designed to be highly portable, instead of taking advantage of all machine instructions on the particular hardware platform. C/C++ compilers typically optimize code for particular processors or classes of processors. Further, JIT technology is very young, compared with other optimization technologies used by FORTRAN and C, so it is probably imperfect.

Good performance with Java code may require careful coding, following guidelines developed and presented here. The alternative is poor performance. For example, if we compared the variant A Java benchmark with the variant B C++ benchmark, then Java would appear to take about 700% longer than C++ (best cases).

More complex Java software involves additional performance issues, such as Java code's tendency to use heap dynamic memory allocation more heavily than C/C++. For example, C/C++ can allocate many objects on the stack, whereas Java must allocate all objects on the heap. Stack allocation receives greater hardware support on modern processors and is more efficient when it is possible. Also, though the frequency of garbage collections can be reduced by careful coding, Java code execution inevitably involves periodic temporary halts to allow the garbage collector to operate.

Java also lacks adequate support for multidimensional arrays and complex numbers. According to Moreira et al. [2], "true rectangular multidimensional arrays are the most important data structures for scientific and engineering computing." In Java, each reference to a *d*-dimensional array involves *d* null pointer checks and *d* subscript range checks. We should also note that Java technology does not use inlining as efficiently as C++.

Coding for efficiency can be much more important for Java than for C/C++. Supporting data comes from Moreira at al. [2]. Table 2 presents their results on the relative speed of two large-scale, computationally intensive benchmarks with several levels of coding alterations to improve efficiency. They implemented three enhancements:

- A vastly improved multidimensional array package for Java (implemented as 100% Java, which manipulates the arrays in a FORTRAN-like manner).
- An improved array package for complex numbers.
- Heavy source-code inlining. Larger numbers indicate better performance. Numerical data are relative speeds (not times).

Table 2. Relative Speed for Two Benchmarks

Deployment	Data Mining Benchmark (relative speed)	Electromagnetic Benchmark (relative speed)
FORTRAN	120	139.5
Java with improved array package and inlining	–	75.8
Java with inlining	–	45.6
Java with improved array package	109.2	0.7
"Plain" Java	25.8	1.2

Through manipulations of the Java source code alone, Moreira et al. were able to obtain Java performance within a factor of 2 of FORTRAN. That result gives us hope that careful coding in Java can lead to adequate performance.

Additional analysis clarifies the problem. Consider the summing of one million 64-bit floating-point numbers. In Java, the *double*s can be stored as a vector (one of Java's standard containers, analogous to C++ containers), an array of objects, an array of *double*s (a Java wrapper class for primitive *double*s), or an array of primitive *double*s. Table 3 illustrates the performance of Java 1.3 code on a 700-MHz AMD K7 platform, with C++ execution times for comparison.

Superficially, Java's container classes seem to be fairly efficient, compared with Java primitive arrays, provided a JIT is used. However, all Java implementations do very poorly relative to C++. An important note: our Java *vector* benchmarks were implemented with *get*(), rather than *iterator*()—because, surprisingly, *iterator*() is slower. In contrast, C++/STL iterators are very fast.

When using Java, two additional factors enter the picture: loop overhead and size counting. To evaluate the former, we unroll the *for* loops as in Table 4. The extension to unrolling at any level (for example, 10×) should be obvious. Run-time results are presented in Table 5.

Table 3. Summation Run Times for Several Implementations and Execution Environments

Implementation	Execution Time without JIT (ms)	Execution Time with Client Hotspot JIT (ms)	Execution Time, C++ (ms)
Vector	1,120	290	20
Object[]	610	260	N/A
Double[]	535	260	N/A
double[]	330	210	24

Table 4. Representative Java Loop Unrolling

Original Code	Unrolled Code (4×)

a. Vector implementation
List data = new Vector(1000000);
...
final int n = data.size( );
double tot = 0;
for (int i = 0; i < n; ++i) {
tot+=((Double)data.get(i)).doubleValue();
}
	b. Vector implementation
	List data = new Vector( );
	...
	final int n = data.size( );
	double tot = 0;
	for(int i = 0; i < n; i += 4) {
	tot+=((Double)data.get(i)).doubleValue( )
	+ ((Double)data.get(i+1)).doubleValue( )
	+ ((Double)data.get(i+2)).doubleValue( )
	+ ((Double)data.get(i+3)).doubleValue( );
	}

c. Array implementation
double[ ] data = new double[1000000];
…
final int n = data.length;
double tot = 0;
for (int i = 0; i < n; ++i) {
tot += data[i];
}
	d. Array implementation
	double[ ] data = new double[1000000];
	...
	final int n = data.length;
	double tot = 0;
	for(int i = 0; i < n; i += 4) {
	tot += data[i] + data[i+1] + data[i+2]
	+ data[i+3];
	}

Table 5. Performance for Summing 10^6 Doubles *with and without Loop Unrolling*

Implementation	Execution Time without JIT (ms)	Execution Time with Client Hotspot JIT (ms)	Execution Time, C++ (ms)
Vector, original, *a*	1,120	290	20
Vector, 4× unrolled, *b*	910	165	14
Vector, 10× unrolled, *b*	850	130	13
double[], original, *c*	330	210	24
double[], 4× unrolled, *d*	200	100	15
double[], 10× unrolled, *d*	130	49	13

Note: *a, b, c,* and *d* refer to Table 4.

We drew the following conclusions:

- Neither the Java compiler nor the JIT appears to unroll loops. Java loop overhead is high, and explicit loop unrolling is helpful. It is disappointing that Java still lacks optimization techniques such as loop unrolling that have been built into FORTRAN compilers for 30 years or more.
- Java vectors are several times slower than arrays in the high-performance case of unrolled loops.
- The best Java *vector* implementation is about ten times slower than C++. The best Java *double* array implementation is about four times slower than C++.

A second well-known loop optimization is removal of constant expressions, including loop limits, as demonstrated in Table 6. Our run-time results are shown in Table 7. Loop optimization is less than perfect, even with the JIT, but that turns out to be a modest effect for vectors and a small effect for arrays. A potentially important issue for Java in or out of loops is *code inlining*. Basically, Java has none today. For example, a standard accessor method is implemented and compiled as in Table 8.

Table 6. Representative Java Loop-Limit Modifications

Original Code	Constant Loop-Limit Code

e. Original code
List data = new Vector(1000000);
...
double tot = 0;
for (int i = 0; i < data.size( ); ++i) {
  tot+=((Double)data.get(i)).doubleValue();
}
	f. Constant loop-limit code
	List data = new Vector(1000000);
	...
	final int n = data.size( );
	double tot = 0;
	for (int i = 0; i < n; ++i) {
	  tot+=((Double)data.get(i)).doubleValue( );
	}

Note: The notation (*e, f*) continues from Table 4.

Table 7. Performance for Summing 10^6 Double*s with and without Loop-Limit Fixing*

Implementation	Execution Time without JIT (ms)	Execution Time with Client Hotspot JIT (ms)	Execution Time, C++ (ms)
Vector, size() in limit, *e*	1,530	320	20
Vector, constant limit, *f*	1,120	290	N/A
double[], 10× unrolled, length operator in limit, *e*	140	49	N/A
double[], 10× unrolled, constant limit, *f*	130	49	13

Note: *e* and *f* refer to Table 6.

Table 8. Java Compilation of Simple Accessor

Java Source Code	Java Byte Code (JDK 1.3)

public class test {
private int value;
public final int getValue( ) {return value;}
public final void run( ) {
final int a = value;
final int b = getValue( );
}
}
	...
	Method void run( )
	0 aload_0 // get object reference
	1 getField #2 // value
	4 istore_1 // a =
	5 aload_0 // get object reference
	6 invokevirtual #3 // getValue( )
	9 istore_2 // b =
	10 return

Java compilers through Java Development Kit (JDK) 1.3 fail to inline even internal references to final definitions of class methods, such as this example. To make matters worse, the Java compiler does not distinguish between final and nonfinal method invocations, which may be overridden through inheritance (though a JIT may optimize differently in the two cases). Run-time results are shown in Table 9.

Table 9. Performance for Summing 10^6 Double*s with and without Accessors*

Implementation	Execution Time without JIT (ms)	Execution Time with Client Hotspot JIT (ms)	Execution Time, C++ (ms)
Vector, 10× unrolled, constant limit, direct data access	925	130	20
Vector, 10× unrolled, constant limit, accessor function	1,000	145	20

Accessors are considered an important, if simple, tool for encapsulation and abstraction in OO development, but developers should be aware that current Java compiler and JIT technology does not optimize them. The implementation of inlining in the JDK appears to have changed with time. Industry reports indicate that some previous JDK versions do inline.

We have confirmed that manipulation of the Java source code can lead to enormous speedups. Specifically, loop unrolling, inlining, and removal of constant expressions from loops can make major improvements in Java performance. Unfortunately, current Java compiler technology does not offer better optimization. However, the problem could be partially resolved by creating a fairly simple preprocessor. Negative side effects resulting from the kinds of source-code manipulation discussed above include

- Increased size of compiled files (Java byte code [JBC])
- More difficult debugging
- Less thorough protection from logic errors, such as overrunning array subscript bounds
- Greater coupling of source code and a resulting need for more recompilation

We believe most side effects can be overcome by performing the optimization in the JIT, rather than at the source-code level (which resolves side effects 1 and 2); and by making the optimization optional and providing optional debugging checks in the optimization process and/or optimizer-generated code (which resolves side effects 2 and 3). Side effect 4 is difficult to overcome. However, the JBC can be optimized, eliminating the need for recompilation—which is what JITs purport to do. Better JIT technology or multipass JBC optimization may provide the best optimization, while avoiding this particular side effect.

Java Performance: Input/Output

To evaluate Java I/O performance, we wrote simple Java and C++ benchmarks that read or write test files to/from memory buffers (byte arrays) with no further processing (using *FileInputStream* and *FileOutputStream*) and with conversion to and from *double*s (using *DataInputStream* and *DataOutputStream*). The tests involved reads/writes of 80 MB on a 700-MHz Pentium III platform, as summarized in Table 10.

Table 10. Java File I/O Performance

Implementation	Execution Time without JIT (s)	Execution Time with Client Hotspot JIT (s)	Execution Time, C++ (s)
Read (80 M raw bytes)	3.6	5.1	4.0
Read (10 M doubles)	505	480	4.0
Write (80 M raw bytes)	4.9	4.6	3.0
Write (10 M doubles)	505	490	3.0

It is remarkable that Java code does about as well as (and occasionally better than) C++ code (using *fread/fwrite*). Equally remarkable is Java's performance about 100 to 150 times worse than C++ performance when converting to/from *double*. The JIT makes little difference. Examination of parts of the *DataInputStream* class source code explains why type conversion is so slow:

```
public final double readDouble() throws IOException {
    return Double.longBitsToDouble(readLong());
    }
public final long readLong() throws IOException {
    InputStream in = this.in;
    return ((long)(readInt()) << 32) + (readInt() & 0xFFFFFFFFL);
    }
public final int readInt() throws IOException {
    InputStream in = this.in;
    int ch1 = in.read();
    int ch2 = in.read();
    int ch3 = in.read();
    int ch4 = in.read();
    if ((ch1 | ch2 | ch3 | ch4) < 0)
       throw new EOFException();
    return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
    }
```

The reading of each *double* item requires five *shift,* four *add*, and one logical *and* operations. Only a very sophisticated optimizer would give good performance with that Java code. For good performance, *DataInputStream*'s methods should have been implemented in native code. Unlike C/C++, Java does not provide a facility for direct, efficient manipulation of individual bytes within primitives. There appears to be no good way to implement *readDouble* in pure Java. Native code appears to be the only reasonable solution.

Another example of this problem is socket I/O. Reading an array of *double*s from a socket stream first as an array of bytes, then as an array of *double*s, we obtain the data in Table 11.

Table 11. Java Socket Read Performance

Implementation	Execution Time without JIT (s)	Execution Time with Client Hotspot JIT (s)	Execution Time, C++ (s)
Socket read: 1,048,576 bytes as raw bytes	0.120	0.125	0.125
Socket read: 1,048,576 bytes as doubles	32.0	29.3	0.125

If an array of *double*s, *float*s, *int*s, or Java primitives other than bytes is read from a socket, the result is the same severe performance penalties associated with the *DataInputStream* or *ObjectInputStream* classes as remarked above for disk I/O. In those cases, the JIT is not very helpful. The Java *DataInputStream, DataOuputStream*, and similar classes should be replaced by native code.

Using Native Code with Java

Java speed may be increased by using "pure" Java for the top levels of logic and non-time-critical parts of the software, then implementing time-critical portions in C/C++ or another language. Java has an extensive system for interfacing with C/C++ executable code or other native machine code—the Java Native Interface (JNI). It facilitates the reading and writing of Java objects from C or C++, the calling of such native methods from Java code, and the calling of Java methods from native code. That capability can be used to facilitate the use of high-performance C/C++-generated machine code.

Indeed, much of the Java infrastructure is implemented in this way. Examples include file and socket I/O systems, which, as we have seen, deliver performance virtually identical to that of C/C++ code. Another example is the Java 3D (J3D) subsystem, in which Java library classes depend heavily on an external OpenGL implementation for their rendering function. J3D classes invoke native methods in a J3D shared library, which interfaces with the external OpenGL system, implemented in native code. As noted below, Java software making effective use of J3D can deliver very high performance, compared with pure Java software.

The JNI must be used carefully if good optimization is to be achieved. In the JNI, both *arguments* and *return values* are copied, so care must be taken that the cost of this copying does not negate the benefits of using native code. An additional difficulty is that native code is, by definition, not very portable. If portability is a requirement, then multiple native code versions would have to be implemented.

Java Performance: Three-Dimensional Graphics

The J3D API, available in implementation from Sun Microsystems as part of the Java platform, is a successful JNI. We began using J3D to solve a data visualization problem requiring manipulation of large spatial data structures containing 100,000 or more points in three dimensions. Requirements included scaling, rotating, and translating/panning the points seamlessly, using both mouse and automated control. Originally, we were concerned that Java would not permit the smooth motion of displayed images in real time. Our early research showed that the Java 2D™ (J2D) API with software rendering could not provide the performance we needed to ensure smooth, real-time three-dimensional displays.

J2D is built on the Abstract Windowing Toolkit (AWT) and does not take full advantage of hardware acceleration. J2D itself, and any attempt to use J2D to animate or render three-dimensional images, depends heavily on the main CPU. J3D is implemented as a mix of pure Java code and JNI calls to OpenGL native implementations. OpenGL, in turn, is supported by numerous readily available graphics accelerator cards (Creative, Guillemot, Hercules, Diamond, ATI Technologies, 3Dlabs, and NVIDIA® Corporation are some of their vendors). The challenge to the developer is to design the software to take maximum advantage of the hardware acceleration that J3D can provide. Indeed, it is possible to write the J3D software in such a way that hardware acceleration of the graphics accelerator cards provides little or no benefit.

We wrote a number of benchmark programs and ran them on a variety of WinTel and Solaris™/SPARC® platforms. Based on our results, we make the following recommendations:

- Fill the J3D data structures with three-dimensional data and implement changes to the data (translation, rotation, and scaling) by operations on the rendering transforms, not by explicit operations on the data. That shifts the computational load from the main

CPU to the graphics accelerator, which can perform the manipulations at much higher speeds. For example, the NVIDIA GeForce 256 boasts 22 million transistors—twice the number in a Pentium III CPU—and is capable of about 25 gigaflops in its geometry engine, whereas current Pentium III processors are capable of only a fraction of a gigaflop.

- Use a graphics card that accurately and efficiently implements the OpenGL API. We have experimented with OpenGL drivers for the Voodoo 3dfx™, NVIDIA-based, and Diamond graphics accelerators. The NVIDIA and Diamond Fire GL1 drivers worked quite well; we never could get the 3dfx OpenGL drivers to work satisfactorily. The Voodoo cards have excellent reputations with PC gamers for high speed, but we found their OpenGL implementations to be incomplete and thus unusable for J3D. Microsoft (Windows) provides a pure software implementation of OpenGL, but we found its performance to be generally poor.

The first set of three-dimensional benchmarks illustrates three different ways to code a 3-D animation that translates an array of points by a fixed amount:

```
// Constructor for the benchmark

public PointBenchmark(int testtype) {
   myPointArray = new PointArray(myNumPoints,
         GeometryArray.COORDINATES|GeometryArray.COLOR_3);
   myPoints = new Point3f[myNumPoints];
   switch (testtype) {
      case 1:
         myPointTransform = new TransformPoints1(myNumPoints, myPointArray,
          0.01f);
         break;
      case 2:
         myPointTransform = new TransformPoints2(myNumPoints, myPointArray,
          0.01f,myPoints);
         break;
      case 3:
         myPointTransform = new TransformPoints3(0.01f, myTransformGroup);
         break;
   }
}
```

The *translateZ*( ) method translates a list of points by a fixed amount in the *z* direction. The method is placed inside the *processStimulus*( ) method, effectively placing *translateZ*( ) inside an infinite loop, and *translateZ*( ) is invoked on every new frame.

```
// The Java 3D behavior scheduler invokes a Behavior node's processStimulus
// method when a ViewPlatform's activation volume intersects a Behavior
// object's scheduling region and all of that behavior's wakeup criteria
// are satisfied.
```

```
private WakeupCondition condition = new WakeupOnElapsedFrames(0);
public void processStimulus(final Enumeration criteria) {
   while (criteria.hasMoreElements()) {
     final WakeupCriterion wakeup = (WakeupCriterion)
criteria.nextElement();
     if (wakeup instanceof WakeupOnElapsedFrames) {
       myPointTransform.translateZ();  //translate points in Z direction
     }
   }
   wakeupOn(condition);
}

public interface ITrans {
   public void translateZ();
}

// This is the slowest way to implement the animation - by adding to every
// point the amount of translation (in the z direction).

public class TransformPoints1 implements ITrans {
   private final Point3f myTemppoint = new Point3f();
   private int myNumPoints;
   private PointArray myStarPointArray = null;
   private float myAmountToTranslate = 0.0f;
   public TransformPoints1(final int numPoints,final PointArray
    pointArray,final float amount){
     myNumPoints = numPoints;
     myStarPointArray = pointArray;
     myAmountToTranslate = amount;
   }
   public void translateZ() {
     for (int l = 0; l < myNumPoints; ++l) {
       myPointArray.getCoordinate(l, myTemppoint);
       myTemppoint.set(myTemppoint.x, myTemppoint.y, myTemppoint.z -
               myAmountToTranslate);
       myPointArray.setCoordinate(l, myTemppoint);
     }
   }
}

// This is a slightly faster way to translate each point by a fixed amount
// in the z direction. Instead of getting each coordinate, calculating the
// new coordinate, and setting the coordinate to the new value, we place //
all the coordinates in a local array of points, modify this array with
// the new point values, and set all the points in the point array at once.

public class TransformPoints2 implements ITrans {
     private final Point3f myTemppoint = new Point3f();
     private int myNumPoints;
     private PointArray myPointArray = null;
```

```
    private float myAmountToTranslate = 0.0f;
    private Point3f myPoints[] = null;
    public TransformPoints2(final int numPoints,final PointArray
        pointArray,final float amount,final Point3f[] points){
      myNumPoints = numPoints;
      myPointArray = pointArray;
      myAmountToTranslate = amount;
      myPoints = points;
    }
    public void translateZ() {
      for (int i=0; i<myNumPoints; ++i) {
        myPoints[i].z -= myAmountToTranslate;
      }
      myPointArray.setCoordinates(0, myPoints); // set the points in the
                                                // point array
    }

  }

// This is the fastest way to translate all the points by a fixed amount in
// the z direction. Once all the points have been set, the only thing
// required to translate the points is to set TransformGroup.setTransform()
// to the correct translation transform.

public class TransformPoints3 implements ITrans {
  private final Vector3f myVector = new Vector3f();
  private final Transform3D myTransform = new Transform3D();
  private float myAmountToTranslate = 0.0f;
  private final TransformGroup myTransformGroup;
  public TransformPoints3 (final float amount, final TransformGroup
       transformGroup){
    myAmountToTranslate = amount;
    myTransformGroup = transformGroup;
  }
  public void translateZ() {
    myTransformGroup.getTransform(myTransform);
    myVector.z -= myAmountToTranslate;
    myTransform.setTranslation (myVector);
    myTransformGroup.setTransform (myTransform);
  }

}
```

It is instructive to compare the speed of execution of the three implementations. Additionally, we tested a second J3D benchmark (Spheres) that contains an animated "solar system" of 60 planetary bodies, plus 50,000 stars, with virtually no explicit computation (the animation is performed using a J3D Alpha, an automated timing device), and a third non-Java benchmark (the industry-standard 3DMark™ 2000 benchmark). Results are summarized in Table 12.

Table 12. Graphics Benchmarks — Java 3D and 3DMark[a]

Benchmark	Approx. Cost without Monitor, 2000 ($)	Trans-form-Points1 (fps)	Trans-form-Points2 (fps)	Trans-form-Points3 (fps)	Spheres (fps)	3DMark 2000[b] (unit-less)
Intel Pentium III (450 MHz), ATI RAGE Pro™	600	2.60	7.94	10.80	4.30	–
AMD K6-II (500 MHz), NVIDIA® TNT2 Ultra	700	2.00	12.07	40.37	8.23	1,309
AMD K7 (700 MHz), NVIDIA GeForce 256 Ultra	1,000	5.25	26.60	91.20	33.00	4,541
Intel Pentium III (700 MHz), NVIDIA GeForce 256 Ultra	1,300	5.52	27.90	104.50	36.40	4,978
Intel Pentium III (700 MHz), NVIDIA GeForce II MX	1,200	5.77	27.00	106.60	41.10	4,657
Dual Intel Pentium III (750 MHz), NVIDIA GeForce 256 Ultra	3,300	5.33	22.30	84.20	37.70	3,681
Dual Intel Pentium III (750 MHz), Diamond Fire GL1	4,000	5.71	26.80	45.60	20.60	–
Sun Ultra 60 (dual 450-MHz UltraSPARC®), Creator 3D	15,000	<1[c]	13.40	34.00	11.20	–

a All WinTel platforms used JDK 1.3 and J3D 1.1.3. The Sun platform used JDK 1.3 beta and J3D 1.2. In all cases, larger numbers indicate higher performance.

b This non-Java benchmark program may be obtained from *www.madonion.com*. It requires Windows 98 or 2000. It will not run on Windows NT® or Solaris platforms. Also, the ATI RAGE Pro and Diamond Fire GL1 video accelerator cards delivered inadequate performance or had insufficient memory to support the benchmark.

c The benchmark program ran so slowly on the Sun platform that the test could not be completed in a reasonable time. In all other cases, the test involved 50,000 data points, but a test with 10,000 points on the Sun ran at 0.026 fps.

Our results led to several conclusions:

- A speedup of 20 times or more may be realized, using the graphics hardware to perform the data transformations, rather than performing them in the main CPU.
- Graphics accelerator cards vary enormously in performance and should be chosen carefully. Price does not always indicate performance—the GeForce II MX accelerator card cost $150 and the Diamond Fire GL1 (which performed somewhat less well) cost $1000.
- CPU performance is also important, though less strikingly so than the graphics hardware. The expense of multiprocessor platforms cannot be justified, unless the additional processors can be used effectively for CPU-intensive processes other than graphics.
- High-priced workstations do not necessarily perform better than inexpensive ones. Our best performance was achieved with a GeForce II MX accelerator card ($150) in a Pentium III personal computer (about $1000, omitting the monitor). The most expensive platform, the $15,000 Sun Ultra 60 (dual 450-MHz UltraSPARC® processors with Creator 3D graphics) delivered extremely poor to mediocre performance.

Native Code Compilers

Another approach to improve Java speed is to compile the Java code fully to machine code. JITs do exactly that at run time. However, some Java compilers produce full machine-code executables with no need for run-time manipulations. Instantiation's JOVE™ is one example. Our results show that compiling to native code can produce better results than current JITs (using 700-MHz Athlon/NVIDIA GeForce 256), as shown in Table 13.

Table 13. Native Code Compiler Performance versus Standard Java and C++

Test	JOVE™	Interpretation	Client Hotspot JIT	C++
Summing 1,000,000 doubles, worst code, *e*	150 ms	1,530 ms	320 ms	24 ms
Summing 1,000,000 doubles, best code, *d*	5.12 ms	130 ms	49 ms	13 ms
TransformPoints1	6.6 fps	1.25 fps	5.25 fps	N/A
TransformPoints2	21.5 fps	6.90 fps	26.60 fps	N/A
TransformPoints3	96.8 fps	85.80 fps	91.20 fps	N/A
Spheres	32.0 fps	24.20 fps	33.00 fps	N/A

Note: *d* and *e* refer to Tables 4 and 6, respectively.

JOVE (Version 1.6) can produce native code executables that run faster than the corresponding C++ code. On the other hand, it produced minimal speedup for the J3D benchmarks versus the JIT. A negative aspect of native compilation is build speed. It is not unusual for JOVE to take 100 times *longer* than the JDK compiler *javac* to compile the same code. Modestly complex software systems can take hours of CPU time on even very fast workstations. Part of the difficulty is that JOVE does not produce a library of relocatable object files (and hence has no incremental build capability), but instead recompiles all code each time it is run. Routine development builds would probably not be feasible using JOVE, at its present speed. However, release builds are perfectly plausible using JOVE .

Multithreading

Unlike C++, Java has threading mechanisms built into the language: synchronization, locks, simple explicit thread production, monitoring, and control. Much C++ code is also multithreaded, but the threading mechanisms are system dependent. The major industry standard (POSIX) is not implemented by Windows. With third-party threading software (such as that available from Rogue Wave Software, Inc), some degree of C++ threading portability can be realized. Threading is built into Java and is perfectly portable.

Multithreading Java code is unlikely to improve absolute performance on a single-processor platform, but multithreading can significantly improve the usability of Java software. For example, a Java applet running in a Web browser can interact with a user while simultaneously downloading data from the Web server and supporting additional browser activity.

Further, Java applets and applications can easily be written to avoid so-called modal dialogs that block the software from further execution until the user responds to the dialog. Finally, multithreaded software automatically exploits multiprocessor platforms.

Java threading details are beyond the scope of this paper but are discussed in many readily available books. However, we have found several pitfalls to Java threading. For example, the core Java graphical user interface (GUI) libraries, the AWT and Swing, are not thread-safe. Sun Microsystems acknowledges that limitation and does not indicate that it will ever be eliminated. Threading can easily cause GUIs to malfunction: lock-ups and incorrect rendering are typical results. Therefore, we recommend the following:

- Never synchronize on an AWT, Swing object, or instantiation of user-created classes that inherit from AWT/Swing classes. Unpredictable lockups are likely to result.
- Operations that significantly alter GUI displays should always be placed on the Java GUI event queue (a simple but critical matter). Failure to observe this precaution is likely to result in corrupted displays. Operations that make no changes to displays or only inconsequential changes can generally be executed asynchronously. It is plausible to write Java GUI software with aggressive multithreading, if a "refresh" button or similar mechanism is provided so that the user can correct a corrupted display.
- All multithreaded code, including non-GUI code, should be written to be thread-safe or reasonably so. Problems and techniques are beyond the scope of this paper but are presented in many texts on threading.

J3D is itself highly multithreaded and appears to be fairly thread-safe. We have not observed the need to take any special precautions when multithreading J3D applications.

General Recommendations

With today's Java technology, developers of time-critical software must be aware of the major Java performance issues. Our recommendations for close attention are as follows:

- *Memory.* Java's excellent design enables coders to make heavy use of dynamic memory allocation, but there is a concomitant performance penalty. It pays to remove allocation of temporaries from loops and other frequently executed code whenever possible. Re-using container objects (vectors, lists, tables, content-addressable arrays, and so forth), instead of reallocating them, is helpful. Constructing an object once and using object setters, rather than frequent construction of the object, eliminates unnecessary memory allocation. If object references are set to null when the referenced objects are no longer needed, the Java garbage collector can collect the memory consumed by that object. Also, Java software can demand more memory than might be expected. Performance can often be improved by adding memory to the deployment workstations/PCs and adjusting the JVM heap parameters.
- *Optimization.* Because of the poor optimization of current Java 2 compilers (including JITs), programmers should perform loop unrolling, inlining, and constant expression extraction either manually or through automated tools. An undesirable consequence is more complex and less readable code in either case.
- *J2D versus J3D.* Java 2D is a powerful API for 2D graphics, but it can be very slow. If insufficient performance is achievable with J2D, try switching to Java 3D, even for 2D-only applications. The J3D API is implemented using OpenGL, which is optimized for performance. On a modest computer with a typical gamer's video card, extremely high

performance is achievable with J3D. That is a great plus for Java. Special care should be taken to choose a graphics card that provides a high level of OpenGL performance and has a complete implementation of OpenGL.

- *Primitives and Arrays.* Unlike C++ (STL), the Java container classes cannot directly manipulate primitive types, such as *int*, *float*, *double*, and *byte*. Further, Java's array-handling capabilities are limited. Software projects that require high-performance manipulation of large volumes of primitives would do well to devote extra resources to writing efficient software for the purpose. Some commercial and freeware packages are available, such as JGL from Instantiations.
- *Java Native Interface.* Unavoidable computationally expensive algorithms can be implemented in C/C++ and invoked from Java code through the JNI. The JNI is easy for developers to use but results in less portability, because the C/C++ code must then be built for each supported platform. Additional care is required to avoid excessive copying of data (JNI arguments and return values are always copied) and excessive overhead due to data conversions between C++ and Java data types, especially for container classes.
- *Just-in-Time Compilers.* Native-code compilers are available for Java, but we have not obtained particularly good results from them. JITs generally perform extremely well, however, and will undoubtedly improve with time. The most performance-demanding projects would be well served by evaluating commercial JITs and compilers when the project allows the deployment environment to be so optimized.
- *Java Virtual Machines.* Some JVM versions have quirks that adversely affect performance in unexpected ways. For example, adjusting some configuration parameters enables font handling to be accelerated by factors of hundreds for recent Java 2 JVMs. The J3D text was unacceptably slow, so we rendered all text by copying the J2D text into a J3D raster. That gave us acceptable performance and the text looked much better.
- *Multithreading.* Unlike C++, Java has built-in multithreading language features, making multithreaded code much easier to write. Multithreading can improve software usability and increase absolute performance on single and multiprocessor platforms. Thread-safe code and multithreaded applications that safely use thread-unsafe Java facilities, such as the AWT and Swing, can be written without great difficulty, if a few basic rules are followed.

Summary

The challenges for the Java developer are to identify which language constructs are fast and which slow, to identify the time-critical portions of software, and to carefully optimize them. Careful coding optimization and "perversion" of the OO structure of the code, as by Moreira et al., may appear to be counter to mainstream thinking about good software practice. We can hope that Java compiler and run-time technology will eventually improve, so that less deviation from the straight-and-narrow OO development path will be necessary for good performance. Modifications to the Java language, such as the creation of purely organizational data types, may be necessary before programmers can be truly oblivious to the problem.

References

[1] J. Byous, "Java Technology Pays Positively," Internet paper available from Sun Microsystems at *http://java.sun.com/features/1998/efficiency.html* (1998).

[2] J.E. Moreira et al., "Java Programming for High-Performance Numerical Computing," *IBM Systems Journal*, Vol. 39, No. 1, 2000, pp. 21–56.

James C. Schatzman is a senior software engineer with TRW Systems in Aurora, Colorado. Specializing in signal and image processing for 17 years, he has been a team lead in software architecture Independent Research and Development (IR&D) projects for several years and in a recent Java technology transfer effort on a large software program. He holds a BS in mathematics and physics from California State University, Fullerton; an MS in computer science from the University of California, Los Angeles; and a PhD in applied mathematics from the California Institute of Technology.

e-mail: James. Schatzman@trw.com

H. Roy Donehower is a software engineer with TRW Systems in Aurora, Colorado. He is currently working on several IR&D projects using Java. He holds a BS from the U.S. Air Force Academy and an MS from the U.S. Air Force Institute of Technology, both in computer engineering.

e-mail: Roy.Donehower@trw.com