Experiences in porting a Virtual Reality system to Java

Shaun Bangay
Computer Science Department
Rhodes University

Abstract

Practical experience in porting a large virtual reality system from C/C++ to Java indicates that porting this type of real-time application is both feasible, and has several merits. The ability to transfer objects in space and time allows useful facilities such as distributed agent support and persistence to be added. Reflection and type comparisons allow flexible manipulations of objects of different types at run-time. Native calls and native code compilation reduce or remove the overhead of interpreting code.

Problems encountered include difficulty in achieving cross-platform code portability, limitations of the networking libraries in Java, and clumsy coding practices forced by the language.

CR Categories: D.3.3 [Programming Languages]: Language Constructs and Features—Frameworks;I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

Keywords: java,networking,serialization,native calls

1 Introduction

The promises of Java include automatic cross platform portability, automatic memory management and networking services that are simpler and better integrated[10]. These factors are offset when constructing real time applications by overheads of code interpretation, and garbage collection. Practical experience from rewriting an existing virtual reality system in Java yields some fresh insights into these issues; some in stark contrast to that which may be expected, either intuitively or from marketing hype.

The goal of this paper is to report on results obtained when translating a large virtual reality system from C++ to Java. The real-time performance requirements of this system differ from the web-based applications for which Java is frequently advocated. The use of a Java system as a computational and data-flow platform, which must interface with a range of devices and peripherals differs from most current applications of the language. The system described also makes extensive use of open source products, both as support libraries and as potential Java platforms; and the implications of this aspect of the system are discussed.

Some benchmarks of different Java platforms are provided, which measure the ability of different systems to provide the facilities required by the virtual reality system, and reflect on the ability of Java code to perform consistently.

2 Related Work

Benchmarks for different Java systems are readily available[5]. The overheads of various constructs such as threads[8] and synchronization[7, 6], native code calls[1] and garbage collection are well documented.

Multimedia applications specializing in 3D computer graphics have been written in Java. Many of these are written purely in Java, to be platform independent and capable of running in Web browsers (Anfy3D `http://www.anfyteam.com/panfy3d.html`, X-Insight `http://www.bssi-tt.com/`, Easy3D `http://www.easy3d.com/`, VisAD `http://www.ssec.wisc.edu/~billh/visad.html`), which can limit performance, and the ability to fully utilize any advanced hardware. The Bang game engine (`http://www.in-orbit.net/open.html`) originally written in Java has moved to C++ purportedly for performance reasons. Java3D is employed in some situations to provide rapid rendering via calls to libraries such as OpenGL or DirectX.

3 The Virtual Reality System

The RhoVeR virtual reality system has been an ongoing development since 1995. The system models a virtual environment and provides support for manipulating the model, rendering the environment and controlling interaction through a range of peripheral devices. Originally developed in C as a system of distributed processes[2], it was soon clear that the complexity of the inter-process communication affected the performance and stability of the system adversely.

The system has been redesigned and implemented in C++ (RhoVeR release **CoRgi**) as an object-oriented system structured around the concept of data flow between components for much of the inter-object interaction. The system is used primarily on MIPS/IRIX and Intel/Linux platforms, although ports to Sparc/Solaris and Intel/Windows2000 have been accomplished without significant difficulty.

The base class of all objects is the `Component`. This class takes care of all scheduling issues, and streaming of data between the different components. Previous versions of RhoVeR provide a separate thread of execution for each component. This has been found to be unreliable, race conditions are difficult to identify and track down; and inefficient, permitting busy waiting to be employed. The `Component` class instead provides a round-robin non-preemptive scheduler which allows a degree of pseudo-concurrency, and is capable of being implemented without ensuring that all classes are completely thread safe. Each component requiring some processor time must provide a method containing the tasks to be performed during one iteration of the system. Care must be taken to ensure that this method returns as soon as possible, and never blocks.

Each component includes a number of `Port` objects. These can be connected to `Port` objects on other components to create a route for the flow of data between components. Components can thus be classified as sources, filters or sinks depending on their position in the data flow. Sources and sinks are associated with device objects

which provide the system specific code for reading from, or writing to hardware devices. Data can be made to flow transparently over a network, by the inclusion of network components in the data flow.

4 Translation to Java

Rewriting the system in Java (RhoVeR release **Dane**) involves a direct translation of the C++ code into Java. The languages are very similar syntactically, and this can be accomplished rapidly.

The Java implementations to which the system is ported are Kaffe-1.0.6 (a clean room Java implementation) under Linux and IRIX, and GCJ-2.96 (a gnu compiler front-end, capable of compiling to native code) under Linux. These platforms are chosen because of their open source licensing which provides greater flexibility when dealing with experimental software. The JDKs provided by IBM (version 1.3.0) and Sun Microsystems (Blackdown version 1.3.0), both for Linux, as well as the JDK (version 1.3.0) from Silicon Graphics for IRIX, are also tested. The MIPS/IRIX and Intel/Linux platforms are targeted for the port because of portability restrictions in some of the device driver software. Some of the Java platforms have limited GUI functionality, containing only a restricted implementation of AWT. Since the Qt widget library is already used in CoRgi, the GUI elements built in this way are retained and accessed in the new version using native calls.

The platforms used for the tests described in this, and later sections are:

- SGI Octane, with dual MIPS R10000, 195MHz processors and MXI graphics running IRIX 6.4.
- Intel dual Pentium III, 600MHz processors with Voodoo3 graphics card, running Redhat Linux 7.0, kernel 2.2.16.

Some limitations of the Java syntax are immediately apparent. The lack of reference parameters results in some inelegant code. The example of networking code which retrieves an array of bytes sent on one of the channels between `Port`s is shown below. The return values are both the byte array and the number of the channel involved. Since both are used independently, it makes no sense to combine them into one class, nor is it efficient to retrieve the values with two function calls.

```
int c [] = new int [1];
bytes = ReceiveBytes (c);
channel = c[0];
```

Reference parameters are also missed when making calls to native code libraries. Returning multiple values from these methods is common, but the overhead of creating a new instance of a Java class, accessing and initializing the member variables and managing the memory is sufficiently complex that the original function of the code is then almost completely obscured. Filling in the elements of an array that is created in Java, and passed as a parameter is a less elegant, but preferable alternative.

The lack of in-line operators ([4], Chapter 15) restricts the use of most operators to primitive numeric types. Thus operations using some of the basic geometric types need to be rewritten from something resembling the original mathematical expression:

```
Vector3D normal =
  (pnts[p2] - pnts[p1]) ^
  (pnts[p3] - pnts[p1]);
```

to the convoluted prefix form:

```
Vector3D normal = Vector3D.CrossProd(
Vector3D.Subtract (pnts[p2], pnts[p1]),
Vector3D.Subtract (pnts[p3], pnts[p1]));
```

The version of Kaffe used under IRIX has proved to be unreliable, and unable to pass its own compliance tests. Problems such as the inability to distinguish between positive and negative integer constants are a hindrance to the correct operation of some components (although surprisingly many still manage to operate).

4.1 Rewrite of the Components

The `Component` concept translates easily into Java, with a few key differences.

To provide the scheduling service, the `Component` class must maintain a static list of all active components. In the CoRgi version, the destructor for the component can be relied on to destroy the object, and more importantly, sever its link with the scheduler. In Dane, when the application forgets about a component, a link will always exist within this list and the component will never be garbage collected. Even worse is that the component will continue to be scheduled. An explicit termination routine is required in this case, unlike the situation for most other objects in which finalization by the garbage collector is adequate.

The data to be transmitted to any one port of a CoRgi component must be all of one particular type, specified at the design time of that particular component. In order to provide a generic port mechanism, all types are recast to byte arrays, and the CoRgi components must rely on prior knowledge to restore type information. In Dane, all messages are derived from a common base class which in turn extends `Serializable`. This allows data to be easily serialized into a byte array for transmission between network components. Type information is retained with the object in Java, allowing the object to identify itself once it has reached the destination. Ports are now able to distinguish different types arriving on the same link, and objects of different types can be multiplexed over a single link.

4.2 Rewrite of the Networking Code

To satisfy the scheduling requirements of the components, each device driver must be able to poll the corresponding device and return immediately if there is no data waiting. The network devices are no exception. In CoRgi, this is accomplished through the mechanism of the `select()` system call, which identifies any active sockets or returns immediately if there are none. An equivalent to this call does not exist under Java[7]. Mechanisms that have been suggested to work around this problem include the use of a separate thread for each socket which blocks until data arrives, but allows the principal thread to continue running; or the re-implementation of the networking services[9].

Since the number of threads that can reasonably be supported on most platforms is limited, it makes no sense to limit system scalability by multi-threading the network libraries. The systems tested exhibit a limit of about 1000 idle threads per process, which would severely limit the number of network connections in any large or medium scale system requiring a thread per connection. Even systems that support larger numbers of threads exhibit poor performance as this limit is approached.

CoRgi provides a more complex networking interface than the traditional socket level calls in any case (it allows for replication of data, as in a multicast environment). Thus in Dane, an interface between Java and C++ occurs for the networking functions at this network interface. Non-blocking and polled socket I/O is implemented using the C++ libraries. The Java level code is presented with a `Channel` class which provides the functionality required.

Transmission of objects across the network is a source of considerable complexity in CoRgi. Various byte orders, alignments and sizes are employed for the primitive types on different architectures. In C++ this is addressed by defining one standard into

JNI

```
JNIEXPORT void JNICALL Java_TextureImage_loadData
(JNIEnv* env, jobject obj)
  jclass cls = env->GetObjectClass (obj);
  jfieldID fid; fid = env->GetFieldID(cls, "x",
"I");
  if (fid == 0) { Serious error }
  env->SetIntField (obj, fid, (jint) x);
  jbyteArray arr = env->NewByteArray (x * y *
4);
  jsize len = env->GetArrayLength (arr);
  jbyte * body = env->GetByteArrayElements
(arr, 0);
    copy values into array
  env->ReleaseByteArrayElements (arr, body, 0);
```

CNI

```
void TextureImage::  loadData ()
  x = (jint) xfield;
  arr = JvNewByteArray (x * y * 4);
  jbyte * body = elements (arr);
```

Table 1: Comparison of equivalent code using JNI and CNI

which all objects are converted before transmission over the network, introducing significant overhead.

Java serialization allows the programmer to ignore these problems. All marshalling and conversion issues are taken care of automatically by the compiler. This functionality is thoroughly exploited by the `InputServer` application, a server which collates the latest readings from each virtual reality peripheral device, and repeats them over the network to all interested applications. The rate of transmission can be set so as to trade off the quantity of network traffic generated against the rate and latency associated with each sample. The CoRgi version of this server has never had any difficulty in reaching the maximum transmission levels, even on the slower workstations that are generally delegated to this task. In Dane, using the same hardware, the server can barely maintain the 30 samples per second set as the lower limit. The overhead of serialization has been identified as the culprit in this case.

4.3 Rewrite of the Device Drivers

The device drivers in the system are responsible for sending data to, and receiving data from the various virtual reality peripherals used. In some cases these may call on operating system level drivers, but often they invoke other libraries (for example: speech recognition and synthesis), or access hardware directly for customized devices, or devices that are unsupported at an operating system level. Java is not intended to be used at this level, but specifies a mechanism, JNI, for accessing system libraries.

JNI proves to be suitable for the task of implementing the virtual reality system's device drivers. The original code used in CoRgi is retained, and a Java interface is provided to allow transfer from language to the other.

The GNU Java compiler offers a different approach to interfacing C++ and Java code via a mechanism labeled CNI.

A segment of code illustrating the differences is shown in Table 1. With CNI the Java class structure is almost transparently visible to the C++ code. Access to member variables can be done directly, and encouraging sharing of values between the C++ and Java aspects of the class. Creating and manipulating elements in arrays no longer requires protection mechanisms. The implications of this cross-language transparency extends beyond just access to external libraries. Tasks better suited to a particular language can be now be

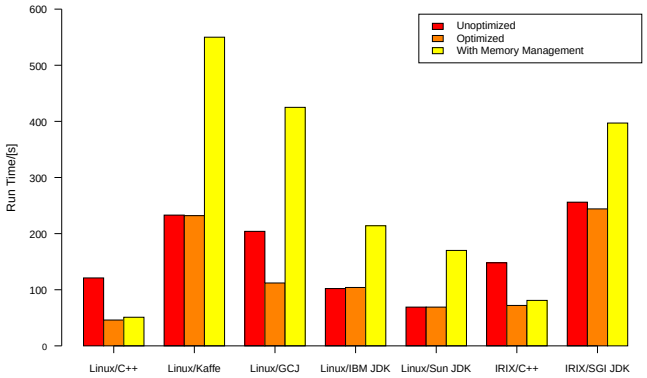


Figure 1: Relative performance for executing code.

written in that language, but still well integrated into the entire system. In our case, the complete functionality of the Qt widget set is now available for providing graphical user interfaces to the virtual reality applications.

5 Conversion Costs

5.1 Platform dependencies

All auxiliary data used by the system is retrieved from a web server, to reduce the number of redundant copies required. The `URL` class turns out to have different semantics in the different systems used. As used in Kaffe and the various JDKs, the header generated by the HTTP protocol is stripped, and the remaining data presented as the contents of the URL. The implementation in GCJ provides both header and data.

As this is a compile time issue, it should be possible to resolve the problem using some form of conditional compilation. This could involve the use of the C pre-processor to conditionally select an appropriate portion of the code to work around the problem.

5.2 Performance issues

While we are not attempting to investigate benchmarks for Java, it is useful to get some comparative measures of the performance of certain operations; particularly those common to the virtual reality system.

5.2.1 Computation Performance

Figure 1 shows the relative performance of the systems used. The test program used calculates points in the Mandelbrot fractal, and gives a measure of some floating point arithmetic and method call overhead. A variation on this dynamically allocates an array to hold some intermediate values, to stress the garbage collector. The equivalent C++ program, with explicit memory management, is also tested.

The difference in performance between unoptimized and optimized code when contrasting the compiled versus interpreted version is interesting, as is the decrease in performance in the case of one system when optimization is turned on. The open source systems perform worst overall, while the original C++ version runs the fastest when optimized. The overhead introduced by garbage collection is significant, suggesting that avoiding dynamic memory allocation, or implementing an explicit memory management system may be a useful approach for real-time systems.

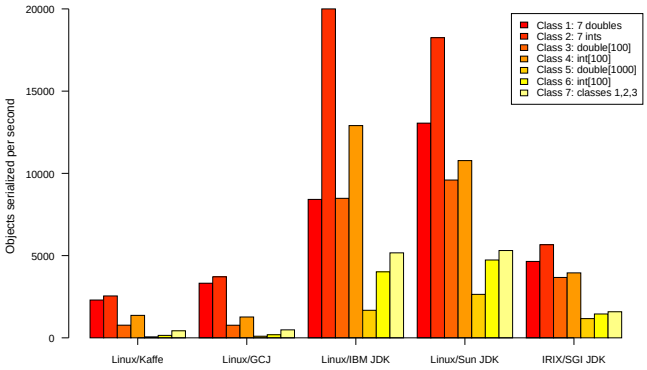


Figure 2: Serialization rates.

5.2.2 Serialization Performance

Directly relevant as a performance measure is the speed with which objects can be serialized. This has already been observed to be a limiting factor in the rate at which device readings can be propagated to client applications. Serialization rates for a range of commonly used structures are shown in Figure 2.

Once again, the open source systems perform badly. It is easy to see how the rate can drop to below 10s of samples per second on slow machines supporting a number of devices. An interesting effect is the greater than tenfold increase in time when increasing the double array by a factor of ten when using Kaffe. Typical device information matches classes 1, 2 and 7, while the larger classes are more typical of agents in the virtual reality system which have softer time constraints.

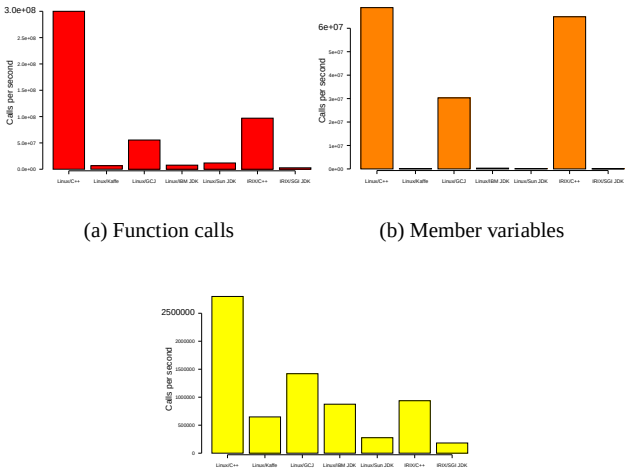
5.2.3 Native call Performance

Rendering, one of the significant aspects of a virtual reality system, is accomplished by making native calls to an OpenGL library. This approach is chosen to simplify porting of the previous version of the system, and because viable alternatives were not available at the time at which translation was started. The performance of native calls are summarized in Figure 3. Calls from C++ to the same functions are shown for reference purposes. CNI is used when profiling the GCJ compiler. The different tests involve: calls to an empty function, calls to increment three double member variables of the Java class and calls to a routine to set the elements of an array of 100 doubles, passed as a parameter.

As may be anticipated, the open source native compiler outperforms the other systems by orders of magnitude, achieving a speed about half that of the pure C++ version. Also interesting is that the other open source solution outperforms the Sun implementation when it comes to array manipulation. The array manipulation test is particularly relevant when passing serialized objects, video images and texture information between the two aspects of the system. Direct manipulation of Java members variables by the C++ native methods allows more complete integration of class methods implemented in the two languages.

5.2.4 Application Performance

The combined effects of the relative performance of the Java platforms used can only be properly assessed in the context of a virtual reality when applied to a virtual reality application. ”Swimming with Dolphins”[3] is a reference application which originated in CoRgi and has been ported with only insignificant changes to the Java version, Dane. The results of measuring the performance of this application are shown in Table 2.



(a) Function calls

(b) Member variables

(c) Array access

Figure 3: Performance of calls to native libraries.

System	Frame rate/ [frames/s]	Rendering /[ms/frame]	Non-rendering /[ms/frame]
Linux/C++	36.27	22.8	4.8
Linux/Kaffe	12.62	70.7	8.6
Linux/GCJ	34.51	25.9	3.1
Linux/IBM JDK	29.29	31.7	2.5
Linux/Sun JDK	15.93	58.7	4.0
IRIX/C++	17.92	42.2	13.7
IRIX/SGI JDK	9.66	96.8	6.7

Table 2: Performance of the “Swimming with Dolphins” virtual reality application.

The highest frame rate for a Java implementation is produced using the GCJ compiler, and comes close to the performance of the original purely C++ system. Rendering time is relatively high compared to the rest of the environment modelling and internal data flow because of the high resolution (1600×1200) used.

Rendering consists of a combination of geometric operations (used in animating the various polygon meshes), and native calls to the OpenGL library to specify transformations and vertex attributes. The relative performance of the different systems during the rendering phase bears a strong resemblance to the results obtained from measurements of the native call overheads, as may be expected.

The rankings for non-rendering overhead is more consistent with those obtained for the computational performance of the various systems. One interesting outlier is the JDK by Sun Microsystems, which performs well in the benchmarks, but fails to live up to expectations when confronted with a substantial application.

5.3 Java enhancements

Use of Java as an underlying platform allows additional functionality to be quickly and easily added to the system.

Virtual reality agents are able to transport themselves between the various virtual reality applications distributed over the different stations running on our network. The agents are themselves objects which support serialization, and can be transmitted by network components between worlds. The current state of the agent is preserved during this process, allowing it to materialize and continue

executing from where it left off. Issues of explicitly identifying this state, and preserving links to other structures used by the object, which would be prohibitively complex in C++ are now automatically taken care of.

Persistent objects can be created by saving the serialized objects to disk. Modifications to the serialization and deserialization process can allow agents which clone (fork) themselves to address a problem in a distributed manner, and then later merge (join) back into a single object with the combined results. Such an approach has already been successful in a distributed messaging system.

Reflection allows the de-multiplexing of the data flow to be performed automatically. Under CoRgi, each C++ component is required to provide a service routine for each port, which could handle the data type received on the port. The Java implementation collates all port handling operations in a parent class, which uses reflection to extract the names of the data types received, and calls a separate event handler appropriate to each type. Implementation of new components no longer requires understanding the mechanisms of the `Port` class, and the precautions required to prevent the service routines from blocking. Instead, if processing of a specific message is required, a handler may be implemented, which deals with a single instance of the message type.

6 Conclusions

In conclusion, the following myths about Java can be debunked:

1. Portability: Java applications are not automatically portable. Practice differs widely from theory when it comes to providing compliant compilers on all platforms, with consistent performance characteristics.

2. Memory management: Automatic memory management, and the pointer/reference model adopted by Java simplifies code creation, while garbage collection has not been found to affect virtual reality applications adversely.

3. Networking: The provision of networking services that are only capable of blocking calls is not appropriate in complex real time applications.

4. Code interpretation: Any overhead with interpretation can be resolved using a native language compiler.

Certain facilities provided by Java are of significant value in expanding the facilities offered by the virtual reality toolkit:

1. Serialization: The ability to load and save objects to byte arrays, and to files allows support for agents, persistence and distribution of the system in ways which are extremely complex using C or C++.

2. Reflection: This mechanism finds use in implementing a simple front-end to the data-flow model used in the system. It also has application in allowing classes (such as agents) to be dynamically added to a running system.

3. Native calls: These are essential for control of devices and integration of legacy software libraries. The CNI approach is far easier to use than JNI, and allows better integration of C++ classes with the Java system.

The integration between C++ and Java provided by the GNU compiler offers the potential of exploiting the best features of both languages simultaneously. A virtual reality toolkit can operate almost as efficiently in Java as in C++, while benefit from the additional facilities offered by the Java architecture.

References

[1] Jack Andrews, Interfacing Java with Native Code, available via the WWW at `http://www.str.com.au/jnibench/`, 2001.

[2] Shaun Bangay, James Gain, Greg Watkins and Kevan Watkins, RhoVeR: Building the Second Generation of Parallel/Distributed Virtual Reality Systems, *First Eurographics Workshop on Parallel Graphics & Visualization*, Bristol(UK), 26-27 September 1996.

[3] Shaun Bangay and Louise Preston, An Investigation into Factors influencing Immersion in Interactive Virtual Reality Environments, In G. Riva, B. Wiederhold and E. Moltinari (Eds.), *Virtual environments in clinical psychology and neuroscience: Methods and techniques in advanced patient-therapist interaction*, Vol. 58, pp. 43-51, Amsterdam, Netherlands: IOS Press, 1998.

[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, The Java Language Specification, Second Edition, available via the WWW at `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`, 2000.

[5] Jonathan Hardwick, Java Microbenchmarks, available via the WWW at `http://www.cs.cmu.edu/~jch/java/benchmarks.html`, March 1998.

[6] Jonathan Hardwick, Optimizing Java for Speed, available via the WWW at `http://www.cs.cmu.edu/~jch/java/speed.html`, March 1998.

[7] Dan Kegel, The C10K problem : Java Issues, available via the WWW at `http://www.kegel.com/c10k.html#java`, February 2001.

[8] Scott Plamondon, The need for speed, stability, available via the WWW at `http://www.javaworld.com/jw-10-1999/jw-10-volano_p.html`, April 2001.

[9] Mark Reinhold, JSR #000051 New I/O APIs for the JavaTM Platform, available via the WWW at `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_051_ioapis.html`, February 2000.

[10] Sun Microsystems, The Source for Java Technology, `http://java.sun.com/`, 2001.