

Performance Issues for Multi-language Java Applications

Paul Murray¹, Todd Smith¹, Suresh Srinivas¹, and Matthias Jacob²

¹ Silicon Graphics, Inc., Mountain View, CA
{pmurray, tsmith, ssuresh}@sgi.com
<http://www.sgi.com>

² Princeton University, Princeton, NJ
mjacob@cs.princeton.edu
<http://www.cs.princeton.edu>

Abstract. The Java programming environment is increasingly being used to build large-scale multi-language applications. Whether these applications combine Java with other languages for legacy reasons, to address performance concerns, or to add Java functionality to preexisting server environments, they require correct and efficient native interfaces. This paper examines current native interface implementations, presents performance results, and discusses performance improvements in our IRIX Java Virtual Machine and Just-In-Time Compiler that have sped up native interfacing by significant factors over previous releases.

1 Introduction

The Java programming environment [1] allows fast and convenient development of very reliable and portable software written in 100% Java. However, in recent years, it has increasingly been used to build large-scale multi-language applications, a trend which is due to several factors.

In many circumstances Java has to interface with existing legacy code written in C/C++. Important examples of this include Java bindings for OpenGL [2], Win32, VIA [3], and MPI [4]. In some performance-critical situations, such as oil and gas applications or visual simulations, application developers have to write core portions of their software in higher performance languages such as C, C++, or Fortran. Finally, developers of server environments need to be able to embed a Java Virtual Machine into servers written in other languages in order to provide Java functionality; examples of this include Java Servlets in the Apache Web Server and Java Datablades in the Informix Database. All these types of applications raise issues of correctness and efficiency in native interface implementations that must be addressed.

This paper is a result of our experiences, as VM and JIT compiler implementors working with our customers, with the effects of interfacing Java with other languages such as C, C++, and Fortran. In Section 2 of the paper we go into detail about the various choices that designers of multi-language Java applications have and which ones are suitable for what purposes. In Section 3 we examine

performance of current Java Native Interface (JNI) [5] implementations. Section 4 discusses issues surrounding memory management and threading in multi-language applications. In Section 5 we present a Fast JNI implementation in the IRIX JVM and JIT that speeds up native interfacing and multi-threaded data accesses by a factor of 3–4x over what is implemented in the standard JavaSoft reference implementation. In Section 6 we discuss related research and in Section 7 we present our conclusions.

2 Choices for Native Interface

Among the many choices that designers of multi-language Java applications have (Fig. 1), the fundamental differences lie in whether Java can coexist and interact directly with C/C++ object models. While j/Direct and KNI tie them together closely, allowing Java objects to be used in C++ contexts and vice versa, JNI takes the approach of separating Java and other languages in a very clear way, which is much more safe and portable, though not very efficient.

As Fig. 1 indicates, JNI is a popular and portable interface, supported by Sun, their licensees, and also other independent software vendors working on clean-room JVM implementations [6][7]. It also allows embedding a Java Virtual Machine within a C/C++ application. This is useful in the context of large servers that need to allow the development of server plugins in Java. The rest of the paper will only look in depth at JNI and its implementation in JVM's derived from JavaSoft's implementation.

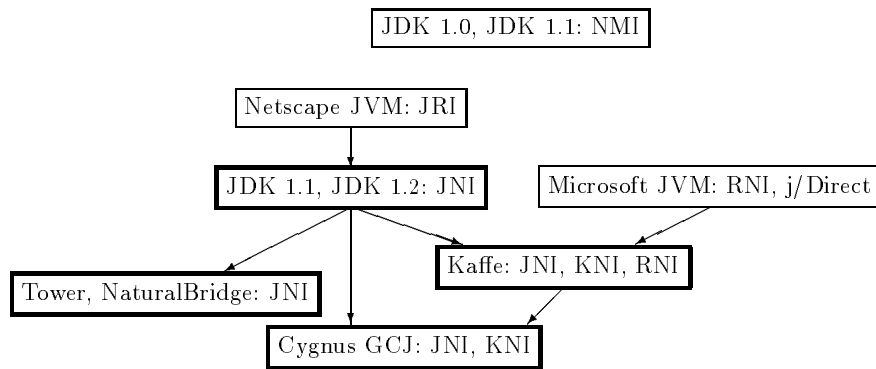


Fig. 1. Choices for interfacing from Java to C/C++. NMI was the original design from Sun and is no longer in active use. Microsoft spawned its own proprietary extensions in RNI (Raw Native Interface) and j/Direct. JNI (Java Native Interface), which evolved from Netscape's work on JRI (Java Runtime Interface), was introduced in JDK 1.1 by Sun and is used in Java2. Clean room implementors such as Tower, NaturalBridge, Kaffe, and GCJ support JNI. Kaffe and GCJ also support KNI (Kaffe Native Interface)

Recommendation 1 *The Java Native Interface (JNI) is a portable and widely supported API for interfacing with C/C++/Fortran codes. Designers should look closely at this before choosing other interfaces.*

3 Java Native Interface Performance

JNI was initially introduced in JavaSoft's JDK 1.1 and the implementations have matured. But there are still wide differences in performance of JNI implementations. This section examines performance results for SGI IRIX and IA-32 Linux and makes recommendations for designers. The SGI tests were performed on an Origin 2000 (300 Mhz R10K) and the Linux test results were obtained on a Dell Pentium II (350 Mhz) running Red Hat Linux. The JVM's used in the tests include SGI's IRIX JDK 1.1.6 [8], SGI's IRIX Java2 [9], Blackdown's [10] Linux JDK 1.1.6, Blackdown's Linux Java2-prev2, and IBM's Linux JDK 1.1.8 [11].

3.1 Cost of a Native Call

One of the most critical pieces of information for developers is the overhead introduced by JNI. One way to determine this is to compare the costs of calling a regular Java method and a JNI method, giving us a rough idea of the overhead introduced for making a native call. In Table 1 we compare the costs of calling an empty Java method and an empty native method one million times.

Table 1. Native call overhead

JVM	Java Method (seconds)	Native Method (seconds)	Slowdown
SGI 1.1.6	41.3	98.0	2.4
SGI 1.1.6 (JIT)	20.8	98.0	4.7
SGI Java2	75.2	76.6	1.0
SGI Java2 (JIT + Fast JNI)	12.7	21.1	1.7
Blackdown 1.1.7	15.0	79.1	5.3
IBM 1.1.8	13.1	81.0	6.2
IBM 1.1.8 (JIT)	0.6*	28.7	47.8*
Blackdown Java2	35.3	49.1	1.4
Blackdown Java2 (Sun JIT)	2.9*	92.8	32.0*
Blackdown Java2 (Inprise JIT)	39.9	49.2	1.2

*The slowdown factors for IBM 1.1.8 and Blackdown Java2 with the Sun JIT are incorrect since the cost of calling a regular Java method is obviously too small; it is likely that JIT optimizations have eliminated the calls of the empty Java method.

From these results we can draw the following conclusions:

- The additional overhead of calling a native method is significant in all cases.
- The cost of calling a native method is lower in Java2 than in JDK 1.1.x.
- If a JIT has specific support for fast JNI calls, they can run significantly faster than under the interpreter (SGI Java2 and IBM 1.1.8); not having such support may cause them to run much slower (Blackdown Java2 with Sun JIT).

Recommendation 2 *For designers of multi-language applications it is important to choose a JVM with support in the JIT for making fast JNI calls, such as IBM's Linux JDK 1.1.8 and SGI's IRIX Java2. In the absence of a JIT, designers should consider newer JVM's such as Java2 over earlier ones like JDK 1.1.x.*

3.2 Cost of Accessing Data from Native Code

Native methods invariably have to access data either in the form of incoming parameters or Java data structures such as arrays. The tests in this section simulate the way that the Java bindings to OpenGL pass around and access data using JNI.

We provide results for several tests that measure data access performance in Table 2. Each one involves calling a native method one million times. In the first one, `scalar`, the method takes 3 float parameters. In the second, `array`, it takes one parameter, a float array of length 3. In the third, `scalar2`, it takes 3 float parameters and then passes them to a helper function. In the fourth, `array-extract`, it takes one parameter, a float array, and extracts it as a C array.

Table 2. Costs of data access from native code. All results given in seconds

JVM	scalar	array	scalar2	array-extract
SGI 1.1.6 (JIT)	128.2	126.7	140.8	410.8
SGI Java2 (JIT)	22.3	31.4	22.3	223.3
IBM 1.1.8	103.7	109.0	105.7	572.7
IBM 1.1.8 (JIT)	29.4	43.7	31.9	511.1
Blackdown 1.1.7	102.3	110.0	107.7	285.8
Blackdown Java2	88.8	53.1	92.2	220.8
Blackdown Java2 (Sun JIT)	106.9	101.5	111.4	286.4
Blackdown Java2 (Inprise JIT)	87.9	53.1	90.7	221.7

When we compare the results to that in the previous section, we see that with support in the JIT, the cost of passing scalar parameters and accessing them is

not significantly higher than calling an empty native method. The results also show that in almost all cases, the cost of passing an array is higher than the cost of passing scalar values. The cost of extracting an array is very high for all the JVM's. In general, Java2 implementations perform better than the 1.1.x implementations, and IBM and SGI outperform Blackdown.

Recommendation 3 *Designers of multi-language applications can build native implementations and use scalar parameter passing and expect reasonable performance when there is JIT and JVM support for fast native interface calling. If they need to pass and extract arrays with the current JVM's, they should be aware of significant costs and should try to optimize by caching and reusing data extracted from the arrays.*

4 Embedding the JVM in Servers: Memory Management and Threading Interactions

One of the big advantages of programming in 100% Java is that all the memory management is fully handled by the JVM and is largely a black box as far as programmers are concerned. The programmer has no real control over Java object locations and lifetimes, and garbage collectors can choose to move objects around. In current JVM's the Java heap and the native heap are in separate areas in the process address space. Crossing from one to the other usually requires copying of data. Although JNI provides a mechanism to access data within the Java heap directly through the `GetPrimitiveArrayCritical` call, the VM implementation decides whether or not a copy is required.

JNI currently does not provide a way to allocate the heap at a specific address, which may be desired by large servers with embedded JVM's, where the Java heap is only one segment of the total memory the server must manage. In general, the cost of native heap allocation in the presence of a JVM is higher than in single-threaded C code since it is done in a thread-safe manner. Finally, the safety of the JVM can be compromised by native routines writing to memory areas they do not own, such as the JVM data structures.

Large servers often already have a threading model, necessitating some complex cooperation between the server code and an embedded JVM. JNI currently has no way of knowing any details about threads created by the server code. This may be necessary, if for example these threads need to use JNI, if the garbage collector needs to scan their C stacks for object references, or if the JVM needs to accurately handle and deliver signals to its own threads as well as server threads.

Recommendation 4 *Embedding a Java VM in an industrial-strength server application needs much more support in both JNI and the JVM than is currently present. Designers should exercise caution before embarking on such Grande applications with current JVM technologies.*

5 Implementation of Fast JNI on IRIX

5.1 Fast JNI Calling Optimization in the MIPS JIT

Sun has defined a JIT API which provides the services that a JIT compiler needs from the VM, including calling JNI methods. However, using it introduces two layers of overhead, translating from the JIT calling convention to the JVM's standard calling convention and then to the JNI calling convention, with a fair amount of other machinery along the way. In all, the journey from a JIT-compiled method to a native method contains three or four levels of function calls, setting up two stack frames, and moving the method arguments around among the registers and memory two or three times. Obviously this has a negative impact on performance, particularly for native methods which are small, such as those in the `java.lang.Math` library.

Since our JIT calling convention and the JNI calling convention are actually very similar, we were able to write an optimized version of this path which handles the vast majority of cases, including all the native methods in the standard libraries. In these cases, the journey from JIT-compiled method to native method now requires one function call, setting up two stack frames, and one fairly quick rearrangement of the method arguments within registers. The benefits of this can be seen in the performance results in Table 3.

Table 3. Speedup from fast JNI calling optimization

Test	Java2 (seconds)	Java2 + Fast JNI (seconds)	Speedup from Fast JNI
Empty Method	69.3	21.1	3.1
scalar	104.2	22.3	4.5
array	74.0	31.4	2.4
scalar2	110.0	23.1	4.8
array-extract	290.7	223.3	1.3

5.2 JNI Pinning Lock Optimization in the JVM

Prowess, a multi-threaded Java oil and gas application from Landmark Graphics, showed poor scalability running on multiprocessor IRIX. Using the Speedshop performance tools and a tracing-enabled POSIX threads library, we found the primary problem to be contention for the JNI pinning lock as the number of threads increased. The JNI pinning lock is held in the reference implementation of the JNI `GetPrimitiveArrayCritical` call while adding the object to a global hash table. By changing the implementation to a lock-per-bucket strategy this contention was largely avoided, resulting in a very large performance improvement, shown in Table 4.

Table 4. Prowess throughput with JNI pinning lock optimization

Threads	Single Lock Throughput (MBytes/sec)	Lock-per-Bucket Throughput (MBytes/sec)	Speedup
2	10.0	70.0	7.0
8	6.5	35.0	5.4

6 Related Work

IBM's JVM team has also developed JNI optimizations [12] similar to those we have implemented in SGI's Java2 JVM. Researchers have developed a variety of solutions for reducing the overhead of interfacing Java to low-level native code. They include extending the Java runtime with new primitives that are inlined at JIT compilation time [3], providing a new type of Java object whose storage is outside the Java heap [3], and introducing a new buffer class and extra garbage collector features that eliminate the need for extra copies [13].

Dennis Gannon et. al. [14] at Indiana University have studied the problem of object-interoperability in the context of RMI and Java/HPC++. Other researchers [15][16] are building tools to automatically create Java bindings to high-performance standard libraries.

7 Conclusions

The lessons we have learned from experience working with our customers in interfacing Java with other languages include:

- While JVM's are beginning to bridge the gap in cost between native method calls and Java method calls, it still deserves careful attention from developers.
- JNI and current JVM's provide insufficient support for industrial server-side applications that want to embed a JVM.
- Our IRIX/MIPS Fast JNI implementation has given substantial performance improvement over previous releases and has improved the performance of real-world applications such as Magician [2] and Prowess.

Acknowledgements

We would like to thank Alligator Descartes of Arcana Technologies [2] for providing us with the benchmark programs used in the performance section. We would like to thank Brian Sumner, an SGI Apps consulting engineer, for working on the Landmark Graphics Prowess application and identifying various performance issues with it. We would like to thank the engineers within SGI working on the Netscape Enterprise Server and at Informix working on database servers for helping us understand some of the issues surrounding embedding JVM's in servers.

References

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
2. Magician: Java Bindings for OpenGL.
<http://arcana.symbolstone.org/products/magician/index.html>.
3. Matt Welsh and David Culler. Jaguar: Enabling Efficient Communication and I/O from Java. In *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications*, December 1999.
<http://ninja.cs.berkeley.edu/pubs/pubs.html>.
4. Glenn Judd, Mark Clement, Quinn Snell, and Vladimir Getov. Design Issues for Efficient Implementation of MPI in Java. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.
5. Sun Microsystems, Inc. *Java Native Interface Specification*.
<http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html>.
6. NaturalBridge. *BulletTrainTM Optimizing Compiler and Runtime for JVM Bytecodes*. <http://www.naturalbridge.com/>.
7. Tower Technologies. *TowerJ3.0: A New Generation Native Java Compiler and Runtime Environment*. <http://www.towerj.com>.
8. SGI JDK 3.1.1 (based on Sun's JDK 1.1.6).
<http://www.sgi.com/Products/Evaluation/#jdk3.1.1>.
9. Java2 Software Development Kit v 1.2.1 for SGI IRIX.
<http://www.sgi.com/developers/devtools/languages/java2.html>.
10. Blackdown JDK port of Sun's Java Developer's Toolkit to Linux.
<http://www.blackdown.org/>.
11. IBM Developer Kit and Runtime Environment for Linux, Java Technology Edition, Version 1.1.8. <http://www.ibm.com/java/jdk/118/linux/index.html>.
12. IBM's Java Technology Presentations.
<http://www.developer.ibm.com/java/jbdays.html>.
13. Chi-Chao Chang and Thorsten von Eicken. Interfacing Java with the Virtual Interface Architecture. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.
14. Dennis Gannon and F. Berg et. al. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10:941–946, 1998.
15. Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-Performance Parallel Programming in Java: Exploiting Native Libraries. In *Proceedings of the ACM 1998 Java Grande Conference*, June 1998.
16. H. Casanova, J. Dongarra, and D. M. Doolin. Java Access to Numeric Libraries. *Concurrency: Practice and Experience*, 9:1279–1291, Nov 1997.