Efficient Cooperation between Java and Native Codes – JNI Performance Benchmark

Dawid Kurzyniec and Vaidy Sunderam

Emory University, Dept. of Math and Computer Science
1784 N. Decatur Rd, Atlanta, GA, 30322, USA
{dawidk,vss}@mathcs.emory.edu

Abstract. Continuously evolving Java technology provides effective solutions for many industrial and scientific computing challenges. These solutions, however, often require cooperation between Java and native languages. It is possible to achieve such interoperability using the Java Native Interface (JNI); however, this facility introduces an overhead which must be considered while developing interface code. This paper presents JNI performance benchmarks for several popular Java Virtual Machine implementations. These may be useful in avoiding certain JNI pitfalls and provide a better understanding of JNI-related performance issues.

1 Introduction

In only a few years Java has evolved from an embedded consumer-electronics programming language to a powerful general-purpose technology used to solve various problems across different hardware platforms. It has already penetrated the enterprise market, is gaining increasing adoption in the field of scientific computing [6,8,2,11], and is even beginning to cope with system level programming and real-time systems.

The evolution of Java technology has eliminated many of the reasons to combine Java with native languages. The performance of modern Java Virtual Machines (VMs) is often able to match pure native code [5]. The number of software components written in Java is growing rapidly, enabling Java to be self-sufficient in most areas. On the other hand, interoperability and reusability of native code simplify smooth migration to Java technology as the value of existing, fine tuned and thoroughly tested native libraries can be retained. Moreover, the wider the area of Java applications, the bigger the demand for interoperability. These reasons were the basis for the development of the Java Native Interface (JNI) [9,10]. The JNI is a platform independent interface specification which impose virtually no restriction on the underlying VM implementations. As a tradeoff for portability, however, such an approach makes it impossible for JNI to be as efficient as it would be, if the interface were integrated more tightly with a specific Java VM.

This paper aims to provide a better understanding of JNI-related performance issues that might help developers make more informed decisions during the software design phase. It presents detailed JNI performance analyses for eight different representative Java VM implementations on three popular platforms.[1] Certain JNI pitfalls are highlighted with suggestions for appropriate solutions for them.

[1] The Invocation API, which enables integration of Java code into native applications, is out of scope of this paper.

2 Compared Platforms and Benchmarking Methodology

All benchmarks described in this paper were performed for the total of eight different Java 1.3 implementations, which included:

- **scli** – SUN HotSpot Client 1.3.0-RC for Solaris,
- **ssrv** – SUN HotSpot Server 1.3.0-RC for Solaris,
- **lcli** – SUN HotSpot Client 1.3.0 for Linux,
- **lsrv** – SUN HotSpot Server 1.3.0 for Linux,
- **lcls** – SUN Classic 1.3.0, green threads, nojit for Linux,
- **libm** – IBM 1.3.0, jitc for Linux,
- **wcli** – SUN HotSpot Client 1.3.0-C for Win32,
- **wibm** – IBM 1.3.0, jitc for Win32.

For both Linux and Win32, the test platform was a Dell Dimension PC with a PII-450 CPU and 128 MB of RAM. Linux tests were run under RedHat 6.2 Operating System while Win32 tests were run under MS Windows 98. Tests for Solaris were performed on 4-processor Sun Enterprise 450 with 4 UltraSPARC 400MHz CPUs and 1280 MB of RAM, running under control of the SunOS 5.8 Operating System. The benchmarking suite was the same for all VMs. All timings are given in nanoseconds. Although the results for different platforms (Solaris, Linux, Windows) were often similar, they should not be compared directly to each other as numerous factors like the amount of available system memory could affect performance. For each test, the mean of at least eight runs was computed and standard variance of each sample was used to determine accuracy. The number of test iterations varied to assure accuracy of at least 1.5 significant digits and was typically between 10^5 and 10^7. In order to give the VM the opportunity to optimize the code, each benchmark was started after a "prime" run with the same number of iterations. Java VMs have very sophisticated run-time optimization algorithms and therefore, the execution times for atomic operations depend on many different factors and cannot be determined precisely. Nevertheless, we believe that our results are highly representative as JNI functions are much less dependent on JIT optimizations than ordinary Java code.

3 Native Method Invocation Overhead

To be used from within a Java application, native code must have the form of `native` methods defined in ordinary Java classes. So the first issue when considering JNI performance concerns the overhead of calling native methods (implemented in separate dynamically linked libraries) in comparison to invoking ordinary methods which may be JIT-compiled and inlined. Performance results measuring that overhead are shown in Table 1. There are separate results for ordinary and `native` methods, for `non-virtual` and `virtual` invocation modes, as well as for methods with no arguments and with eight arguments of type `Object`. For `virtual` invocation mode, methods were invoked through a reference to the superclass to enforce virtual method dispatch. For `non-virtual` invocation mode, the invoked methods were `private`. Fields marked with * denote cases when results in the sample varied significantly and tended to concentrate around two distinct values.

Table 1. Java vs Native Method Invocations – times in [ns]

	Solaris		Linux				Windows	
	scli	ssrv	lcli	lsrv	lcls	libm	wcli	wibm
Java, non-virtual, no args	40	0	20	0	260	25	0	36
Java, non-virtual, 8 args	70	0	35	0	390	50	0	50
Java, virtual, no args	90	*90	22	275	340	30	26	38
Java, virtual, 8 args	90	*220	50	285	420	55	45	56
native, non-virtual, no args	110	150	105	120	500	130	140	120
native, non-virtual, 8 args	340	290	210	310	940	225	710	250
native, virtual, no args	120	170	110	100	460	130	160	125
native, virtual, 8 args	400	310	205	300	970	240	720	255

Notable advance in performance can be observed here between JIT-enabled VMs and the old-style **lcls** VM. The modern ones required between 100 and 170 ns per `native` method invocation plus about 15-25 ns per argument conversion, except for the SUN's HotSpot Client VM for Win32 (**scli**) which required as much as about 70 ns per argument conversion. The overall `native` method invocation overhead turned out to be about 3-5 times bigger than for ordinary methods, but is worth noticing that in some cases JIT compiler was able to eliminate the latter completely.

4 Callback Method Invocations

In many nontrivial Java to native code interfaces, callbacks may play an important role causing Java methods to be called from native side. The results shown in Table 2 focus on the performance of such callbacks and cover invocations of `virtual`, `private` and `static` methods having no arguments or eight arguments, respectively. These results do not include the time needed to obtain method and class identifiers needed prior to invoking the method through JNI (since they need to be obtained only once) nor do they include JNI exception checking code. This test uncovered significant differences between the compared VMs. Only three of JIT-optimized VMs (**libm**, **wcli** and **wibm**) demonstrated acceptable performance requiring between 800 and 1350 ns per method call plus 25-50 ns per each passed argument. SUN HotSpot VMs for Solaris and Linux performed poorly requiring 2500-9000 ns per call (demonstrating especially high overhead for `virtual` invocation mode) and as much as about 900 ns per each passed argument.

5 Field Access

Table 3 presents the performance results for Java field accesses from JNI. There were three separate JNI benchmarks: in the first one, the instance field of the same object to which the native method belonged was accessed; the second shows results for accessing the instance field of another object; and the third benchmark shows access times of a static field. In addition, a separate experiment was conducted to determine the average field access overhead in pure Java. As before, the time for

Table 2. Method invocations from JNI – times in [ns]

	Solaris		Linux				Windows	
	scli	ssrv	lcli	lsrv	lcls	libm	wcli	wibm
private, no args	3500	2500	4100	4000	1300	1100	900	1200
private, 8 args	10100	9500	9100	9000	1800	1500	1250	1500
virtual, no args	9000	6500	9500	8600	1350	1350	830	1200
virtual, 8 args	15500	14000	14000	14400	1500	1550	1050	1600
static, no args	3100	3100	4500	4400	1200	1100	800	1100
static, 8 args	9900	9500	9100	9900	1700	1500	1170	1400

acquiring field and class identifiers was not included. The same three VMs: **libm**, **wcli**, and **wibm** presented the best performance requiring only about 110-140 ns per field access in contrast to 190-290 ns of Linux HotSpot VMs and 470-650 ns of Solaris ones. Notice that field access overhead turned out to be an order of magnitude smaller than those of callback method invocations.

Table 3. Field access from JNI – times in [ns]

	Solaris		Linux				Windows	
	scli	ssrv	lcli	lsrv	lcls	libm	wcli	wibm
JNI, own	590	500	260	260	190	120	110	120
JNI, other's	470	460	285	275	200	120	110	130
JNI, static	650	610	290	290	180	120	140	110
Java	20	20	<10	0	40	0	<10	<5

6 Arrays and Strings

Perhaps the most important data structures in high performance computing are plain, large arrays of primitive types. As currently Java is considered appropriate for high performance computing, and because the demand for interoperability is very strong in this matter, it becomes crucial that Java arrays could be efficiently accessed from within native side. JNI offers three distinct ways to access Java arrays of primitive types. Using `Get / Release<type>ArrayContents()` routines is one approach, where the Java array may be manipulated through a directly exposed native style pointer. The problem with this approach, however, is that it is up to the Java VM implementation whether it pins down the array or instead makes a copy of it prior to returning this pointer. In the latter case performance can be degraded and some memory problems may occur for large arrays. The first two rows in Table 4 report on tests of this method for accessing `int[]` arrays of length 100 and 1000000, respectively.

Another method of accessing arrays is using `Get / Set<type>ArrayRegion()` routines, but this is appropriate only when some small and precisely known portion

Table 4. Array and string access from JNI – times in [ns]

	Solaris		Linux				Windows	
	scli	ssrv	lcli	lsrv	lcls	libm	wcli	wibm
array, int, 10e2	5500	5200	3700	3300	1450	310	700	330
array, int, 10e6	3.1e7	3.1e7	8.3e7	9.0e7	1460	300	710	330
array, int, 10e6, critical	590	690	410	330	1470	370	810	410
string, 3	2400	2100	2700	2200	1550	300	620	330
string, 65536	5.3e5	5.3e5	5.0e5	5.3e5	1520	330	610	310
string, 65536, critical	580	580	320	320	1580	310	660	360
string, UTF, 3	2340	2800	3700	3000	5000	1700	1500	1300
string, UTF, 65536	1.5e6	1.5e6	3.2e6	3.2e6	2.6e6	2.3e6	1.6e6	2.0e6

of the array is accessed; therefore this approach was not tested in our benchmark experiments. The third method is to use the `Get` / `ReleaseArrayCritical()` functions. According to the specification [9], this approach is very similar to the first one except that the Java VM is more likely to pin down the array instead of copying it. However, use of these routines is subject to some important restrictions on the enclosed native code semantics [9]. The third row in Table 4 shows the results for this approach. For clarity, results for arrays of types other than `int` are omitted in Table 4 as the tests have shown that there are no important performance differences except for the obvious dependence on element size in cases when whole arrays were copied.

For strings, JNI offers several distinct access methods as well. One is the `Get` / `ReleaseStringChars()` pair, which is similar to `Get/ReleaseArrayContents()`; performance figures are listed in rows 4 and 5 of Table 4 for strings of length 3 and 65536, respectively. The other way is to use `Get` / `ReleaseStringCritical()` functions which are similar in semantics to `Get` / `ReleaseArrayCritical()`. The results for this approach are presented in row 6. The next two rows refer to yet another access method that converts Unicode Java strings on the fly to the UTF-8 [9] format, which is more natural for most native languages as it is consistent with the ASCII character set. It is also possible to get a copy of a string segment with `GetStringRegion()` routine but it has not been included in our tests.

As can be read from the gathered data, the array and string access benchmarks were dominated by IBM's Virtual Machines, since they were able to avoid array copying and required only about 300-400 ns per array or string access. Such performance could be obtained also using HotSpot VMs for Solaris and Linux but only with `Get...Critical()` routines. Interestingly, the disability to perform array pinning on `Get<type>Array()` function calls was discovered not to be a common issue of all HotSpot VMs as the Client VM for Win32 (**wcli**) was able to avoid it as well. The Classic VM for Linux (**lcls**), which lacks for JIT compiler support and it generally less efficient than modern VMs, also managed to to this but with a few times bigger performance overhead. The probable reason why modern VMs had problems with array pinning is that they employ more sophisticated memory management algorithms.

7 Exceptions

Table 5 presents results for exception-related JNI routines. The first row shows the overhead of the `throw` statement in pure Java. The next row illustrates the same overhead when an exception is thrown using JNI. The subsequent two rows present the performance of the `catch` operation performed through JNI in the case when there was no exception thrown (the most common case), and conversely, when there was a pending exception. It was surprising to note that the overheads of the `throw` and `catch` operations differ by almost three degrees of magnitude (18000-80000 ns and 40-740 ns, respectively). Nonetheless, it is reasonable as the `throw` operation is rare in properly written programs so its efficiency can be often sacrificed to improve overall performance. As in several previous tests, the trio of **libm**, **wcli** and **wibm** demonstrated the best performance here needing only 40-65 ns for exception check in the most common case of no pending exception, whereas the same operation took about 100 ns for Linux HotSpot VMs and about 300 ns for Solaris ones.

Table 5. Exception handling from JNI – times in [ns]

	Solaris		Linux				Windows	
	scli	ssrv	lcli	lsrv	lcls	libm	wcli	wibm
Java, throw	22800	19000	37000	29000	5000	12000	9200	12000
JNI, throw	52000	53000	82000	88000	23500	60000	18000	35000
JNI, catch, no exception	300	320	95	110	40	65	40	40
JNI, catch	740	740	350	350	190	180	160	140

8 Miscellaneous JNI Operations

Table 6 shows performance results for several commonly used JNI features which does not fit in the categories outlined so far. In the first two rows, the execution costs of the `synchronized` statement in pure Java and in JNI (which involves usage of `MonitorLock()` and `MonitorUnlock()` operations) are compared. [2] The next two rows compare overheads of small object instance creation in pure Java and JNI (but without constructor invocation). Rows 5 and 6 compare runtime type check overheads in pure Java and JNI (with `IsInstanceOf()` function). In JNI, there might be several distinct reference variables with different values that refer to the same Java object. To determine this the `IsSameObject()` function is used. Rows 7 and 8 in Table 6 show the overheads of this operation in cases when references indeed refer to the same object and when they do not (which is more probable). Finally, the last three rows in Table 6 refer to several other widely used JNI functions used for various purposes, like creating a new reference pointing to given object, getting the class of a given object and the superclass of given class.

As before, three of the tested JIT-enabled VMs: **libm**, **wcli**, and **wibm** performed much better than the others. The HotSpot VMs for Solaris and Linux

[2] The lock was acquired and released by one-threaded application.

Table 6. Remaining benchmarks – times in [ns]

	Solaris		Linux				Windows	
	scli	ssrv	lcli	lsrv	lcls	libm	wcli	wibm
Java, synchronized	190	150	45	50	870	160	270	140
JNI, synchronized	1400	1320	1350	1300	740	240	300	290
Java, `new`	180	0	100	145	1890	560	880	500
JNI, `AllocObject`	1260	1330	1290	1270	1340	590	1000	470
Java, `instanceof`	30	15	17	0	95	<5	27	10
JNI, `instanceof`	410	410	300	160	150	155	130	110
JNI, `IsSameObject`, `true`	1390	1240	670	660	460	320	310	300
JNI, `IsSameObject`, `false`	340	330	130	100	50	70	50	50
JNI, `GetObjectClass`	480	380	280	240	230	150	190	170
JNI, `GetSuperclass`	620	480	510	350	250	80	120	110
JNI, `NewLocalRef`	490	460	270	260	200	100	110	110

confirmed vendors' promises and presented blasting performance in the synchronization and object creation from pure Java; however, the appropriate JNI equivalents of these operations performed not so good. It can be also observed that the `IsSameObject()` routine is optimized for case when compared references do not point to the same object, and it can take as few as 50 ns for the fastest VMs to detect it.

9 Benchmark Summary

As might have been expected, obtaining Java functionality from native code via JNI function calls turned out to be much slower than pure JIT-optimized Java. Nevertheless, the overhead factor rarely exceeded 30 what is acceptable in most cases as JNI functions typically take only a small part in total native method execution time. Therefore, the overall JNI performance seems to be adequate for most applications where it really have to be used; however, there are several issues that one has to be aware of:

- Copying arrays and strings instead of pinning them down can degrade performance. Unfortunately, even `Get...Critical()` routines do not guarantee that copying will be avoided; nevertheless, they seem to be the most efficient way to access Java arrays and strings.
- For native methods with very small amounts of computing, the additional invocation overhead can exceed the performance benefits.
- Intensive callbacks from native methods can be expensive on some Java VMs and should be used with caution.
- As JNI implementations are not the most important parts of Java Virtual Machines, their performance is not necessarily going to improve – in fact it it can happen that a new VM version from the same vendor would perform JNI calls much worse than an older version, as was the case with HotSpot VMs for Linux (**lsrv**, **lcli**), in which JNI implementations are much less efficient than that from the Classic VM (**lcls**).

10 Conclusions and Future Work

This paper focuses on approaches in the creation of efficient Java to native code interfaces. It provides detailed performance benchmarks of several popular, modern, and representative JNI implementations, pointing out their weak points and suggesting possible solutions.

The ***Janet*** [4,3] is a Java language extension and preprocessing tool which enables convenient development of JNI-based interfaces. It completely hides the JNI layer from the user, defining new syntactic constructs which enable mixing native and Java codes directly. The Harness system [11,7] is an experimental metacomputing framework based upon the principle of dynamically reconfigurable distributed virtual machines. The ***Janet*** language extension and experiences gained from collected JNI performance data will be combined to make Harness aware of native code resources and libraries, increasing its interoperability and potential field of applications.

References

1. J. Andrews. Interfacing Java with native code – performance limits. http://www.str.com.au/jnibench/.
2. R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. In *ACM-1998 Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February 1998. Available at http://www.cs.ucsb.edu/conferences/java98/papers/jnt.ps. 1
3. M. Bubak, D. Kurzyniec, and P. Łuszczek. Creating Java to native code interfaces with Janet extension. In M. Bubak, J. Mościnski, and M. Noga, editors, *Proceedings of the First Worldwide SGI Users' Conference*, pages 283–294, Cracow, Poland, October 11-14 2000. ACC-CYFRONET. 10
4. M. Bubak, D. Kurzyniec, and P. Łuszczek. A versatile support for binding native code to Java. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *Proceedings of the HPCN Conference*, pages 373–384, Amsterdam, May 2000. 10
5. O. P. Doederlein. The Java performance report. http://www.javalobby.org/features/jpr/part3.html. 1
6. V. Getov, P. Gray, and V. Sunderam. MPI and Java-MPI: Contrasts and comparisons of low-level communication performance. In *SuperComputing 99*, Portland, USA, November 13-19 1999. 1
7. Harness project home page. http://www.mathcs.emory.edu/harness. 10
8. Java Grande Forum. http://www.javagrande.org. 1
9. Java Native Interface. http://java.sun.com/j2se/1.3/docs/guide/jni/index.html. 1, 6
10. S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999. 1
11. M. Migliardi and V. Sunderam. The Harness metacomputing framework. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, USA, March 22-24 1999. Available at http://www.mathcs.emory.edu/harness/PAPERS/pp99.ps.gz. 1, 10
12. M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12:519–538, Dec. 1999. Special Issue on Java for High-Performance Applications. Available at http://www.cs.berkeley.edu/~mdw/papers/jaguar-journal.ps.gz.