Performance issues for Multi-language Java applications

Paul Murray, Todd Smith, Suresh Srinivas
Silicon Graphics Inc
Mountain View, CA 94040
{pmurray, tsmith, ssuresh}@sgi.com
Matthias Jacob
Princeton University
Princeton, NJ
mjacob@cs.princeton.edu

Abstract

The Java programming environment [1] is increasingly being used to build large-scale multi-language applications. This trend is due to several factors including:

- Legacy libraries being wrapped with a Java interface (e.g., Java bindings for OpenGL [2], Win32, VIA [3], MPI [4], etc.)
- Java applications with C/C++/Fortran cores written to address performance issues (e.g., Oil and Gas applications, visual simulations, etc.).
- Servers embedding Java Virtual Machines that allow the development of server plugins in Java (e.g., Apache Web Server and Servlets, Informix Database and Java Datablades)

There are a number of issues both in terms of correctness and efficiency that need to be worked on to make this really viable. In this paper we identify and discuss in depth the following issues:

- The strengths and weaknesses of several Native Interface implementations.
- The performance implications of interfacing with C/C++ from Java.
- Embedding of the Java VM in servers.

We then present an implementation within our MIPS Just-In-Time compiler [5] that speeds up native interfacing by a factor of 3 to 4 over what is implemented in the standard JavaSoft reference implementation. We also present performance results for interfacing with native code for other platforms such as Linux and Windows NT to help understand the state of the art in native interfacing.

1 Introduction

This paper discusses 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.

Java allows fast and convenient development of very reliable and portable software. But in many circumstances Java has to interface with existing legacy code written in C/C++, and in some situations application developers have to write core portions of their software in higher performance languages such as C. With improvement in Java compilers and VM's the performance of 100% Java programs may continue to approach C/C++ performance. In the meantime designers of large-scale applications have to understand some of the implications of mixing languages such as Java with C/C++.

Some of the big issues facing designers of multi-language Java based applications include:

- What are the choices that application developers have for developing, debugging, and performance tuning multi-language Java applications?
- What are the differences in performance between calling a regular Java method and calling a native method?
- How do the memory management of the Java VM and the memory management of native libraries interact?
- How does one interface servers that are multi-threaded with the Java VM and manage the threading model correctly?

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 purpose. In Section 3 we will look at performance of Java Native Interface (JNI) [6] implementations. Section 4 discusses issues surrounding memory management and threading in multi-language applications. In Section 5 we then present a Fast JNI implementation in the MIPS JIT and IRIX Java VM that speeds up native interfacing and multi-threaded data accesses by significant factors. In Section 6 we discuss related research and in Section 7 we present our conclusions.

2 Choices for Native Interface

Given the many choices that designers of multi-language Java applications have (*Figure 1*), picking the right one and understanding the issues can be quite daunting.

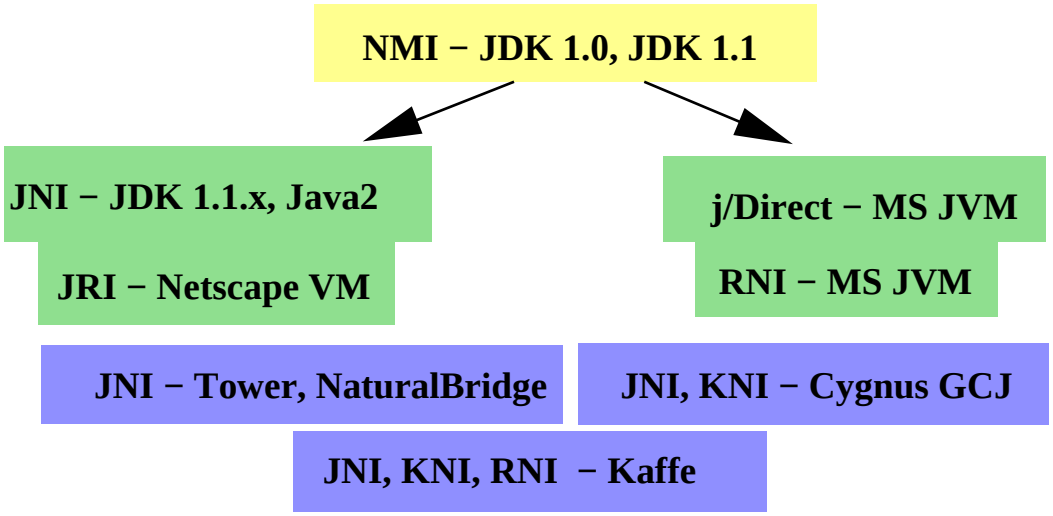

Figure 1: **CHOICES for interfacing from Java to C/C++. NMI was the original design from Sun and is no longer in active use. Microsoft has spawned its own proprietary extension in RNI (Raw Native Interface) and j/Direct which sits on top of RNI and uses annotations to provide the Win32 interface from Java. JNI (Java Native Interface), which evolved from Netscape's work on JRI (Java Runtime Interface), was introduced in Java 1.1 by Sun and is used by the JVM in the Java2 implementation. Clean room implementors such as Tower, Naturalbridge, and GCJ support JNI. Kaffe recently announced support for both JNI and Microsoft's RNI. Kaffe and GCJ also support KNI (Kaffe Native Interface).**

As *Figure 1* indicates, JNI is one of the popular and portable interfaces that is supported by Sun, their licensees, and also other independent software vendors working on clean-room JVM implementations [7][8]. It is important to realize that if a designer's predominant deployment platform is Windows and they need to use

Win32 and Microsoft COM, they should look closely at j/Direct [9] and RNI [10] from Microsoft. If they really want portability then JNI is their main option.

The fundamental differences between many of these API's lies in whether Java can coexist and interact directly with the object models (including structure layouts) of C/C++. j/Direct [9] and KNI [11] try to make Java just another programming language and allow Java objects to be used in C++ contexts and vice versa. In the case of j/Direct, this would be of great convenience to Visual Studio customers using Visual J++ and interfacing with Visual C++. In the case of KNI, it is meant to ease the interaction between g++ and Java.

JNI takes the approach of separating Java and the other language in a very clear way. This is much safer and portable though not very efficient. The JNI API also allows embedding of 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 into the JNI API and its implementation in JVM's derived from JavaSoft's implementation.

The state of the art for debugging and performance tuning multi-language Java based applications is rather primitive. This is a rather fertile area for interesting research work.

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 Performance of Java Native Interface

JNI was initially introduced in JavaSoft's Java Development Kit (JDK) version 1.1 and the implementations have matured. But there are still wide differences in performance of JNI implementations. This section looks at performance results and provides recommendations to designers. In the current version of the paper we provide results for SGI IRIX and IA-32 Linux. 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 JVMs used in the tests include SGI IRIX JDK 116 [12], SGI IRIX Java2 beta [13], Blackdown [14] JDK 116, Blackdown Java2-prev2, IBM's JDK 118 [15]. We will provide results for the NT platform in the final version of the paper.

3.1 Cost of a native call

One of the first pieces of information a designer would like to know is the cost of 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. This gives us a rough idea of the overhead introduced for making a native call. We call an empty Java method 1 million times and compare that to calling an empty native method 1 million times.

JVM	Regular method (secs)	Native Method (secs)	Slowdown (Native/Regular)
SGI 116 (nojit)	41.3	98.0	2.4
SGI 116 (mipsjit)	20.8	98.0	4.7
SGI Java2 (nojit)	75.2	76.6	1.0
SGI Java2 (mipsjit+fastJNI)	12.7	21.1	1.7
Blackdown 117 (nojit)	15.0	79.1	5.3
IBM 118 (nojit)	13.1	81.0	6.2
IBM 118 (jit)	0.6	28.7	47.8 (?)
Blackdown Java2 (nojit)	35.3	49.1	1.4
Blackdown Java2 (sunwjit)	2.9	92.8	32 (?)
Blackdown Java2 (inprise)	39.9	49.2	1.2

From *Table 1* we can draw the following conclusions:

- The added cost of calling a native method over calling a regular Java method is significant in all cases we measured.
- Comparing the costs of calling a native method with and without the JIT shows that SGI Java2 and IBM's 1.1.8 have support in the JIT for making fast JNI calls.

- The cost of calling a native method is lower in Java2 than in JDK 1.1.x and is significantly lower with support in the JIT for fast native access, as can be seen in Linux IBM 1.1.8 and SGI Java2.
- Also, if the JIT does not have support for fast JNI, the overhead it introduces may cause it to run slower than the interpreter, as the results for the sunwjit shows.

The slowdown factors for IBM and sunwjit 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.

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. Linux IBM 1.1.8 and SGI IRIX Java2 are such JVM's. 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 Java bindings to OpenGL use the Java Native Interface.

We provide results for several tests that measure data accesses in *Table 2*. In the first one, Param-scalar, we call a native method taking 3 float parameters 1 million times. In the second, Param-array, we call a native method and pass to it an float array (of length 3) 1 million times. In the third, Param-scalar2, we call a native method taking 3 float parameters 1 million times and within the native method we call a helper function to which the parameters are passed. In the fourth, Param-array-extract, we pass a Java float array and extract it in the native function as a C array.

Test	SGI 116 mipsjit (secs)	SGI Java2 mipsjit (secs)	IBM 118 nojit (secs)	IBM 118 jit secs
Param-scalar	128.2	22.3	103.7	29.4
Param-array	126.7	31.4	109.0	43.7
Param-scalar2	140.8	22.3	105.7	31.9
Param-array-extract	410.8	223.3	572.7	511.1
Test	Blackdown 117 (nojit)	Blackdown Java2 (nojit)	Blackdown Java2 (sunwjit)	Blackdown Java2 (inprise-jit)
Param-scalar	102.3	88.8	106.9	87.9
Param-array	110.0	53.1	101.5	53.1
Param-scalar2	107.7	92.2	111.4	90.7
Param-array-extract	285.8	220.8	286.4	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 (except Blackdown Java2 nojit), the cost of passing an array is always higher than the cost of passing scalar values. Also the cost of extracting an array is very high in all of the JVM's. There are no significant differences between Param-scalar and Param-scalar2 for SGI Java2, probably due to the SGI C compiler optimizing away the empty call. Also Blackdown's JDK is significantly slower than IBM and SGI's for native method calls and parameter accesses. Java2 implementations are considerably better at extracting arrays than the 1.1.x implementations.

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 JVMs they need to be aware of significant costs and should try to optimize by caching and reusing data extracted from the arrays.*

More detailed performance results for data accesses, calling Java code for native code, etc., will be provided in the final paper.

4 Memory Management and Threading interactions

This section documents some of the issues that we as JVM implementors encountered when some of our customers wanted to embed JVM's in their server applications. These applications included commercial databases, commercial web servers and applications that used ISV libraries that have JNI components.

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 (such as copying collectors) can choose to move objects around. In the 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 in Java2 JNI there is a mechanism to get to data within the Java heap directly through the `GetPrimitiveArrayCritical` call, the VM implementation decides whether or not a copy is required. This often presents an efficiency concern for implementors who would like to provide Java interfaces to low level hardware resources such as network interfaces.

The current JNI interface does not provide any way to allocate the heap at a specific address. This becomes necessary when a large server application that is managing all its memory and embedding a JVM needs control over where the Java heap is allocated. The cost of allocation is also higher than in single-threaded C code since allocation is done in a thread-safe manner either by using the thread-safe version in the standard C library or by explicit locking and overriding of the `malloc` and `free` routines by the JVM. Finally the safety of the JVM can be compromised by native routines writing to memory areas they do not own, such as the Java VM data structures. Users also do not have control over enabling tracing or checking for memory management routines or following some of the standard allocation debugging techniques.

Large servers that want to embed a JVM already have a threading model within them, and both the JVM and the server need to cooperate on resources such as signals. The JNI API today does not have any way to know any details about the threads that are created by the server application. In some cases this may be necessary, for example if they use the JNI API, or if the garbage collector needs to scan their C stacks for object references. Also as a result of not knowing about the server threads, the JVM is not equipped to handle server reliability and may trip over system resources such as signals being delivered to the JVM when they were intended for server threads.

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

These topics should be of active interest to working groups in forums such as Java Grande [16], a group which is addressing the growing interest in using Java for high-performance applications.

5 Implementation of Fast JNI on IRIX

5.1 Fast JNI calling optimization in MIPS JIT

The SGI JVM, based on JavaSoft's reference implementation, has the concept of an *invoker*, a standard interface for invoking a method regardless of the way it is implemented, for example in bytecode or in native code. An invoker takes as its arguments an object (or a class in the case of a static method), a method data structure, the number of arguments being passed, and an execution environment data structure, which among other things provides the stack frame of the calling method, which contains the arguments themselves. For each method which has been loaded into a running VM, its method data structure contains a pointer to an appropriate invoker for the method.

For JNI methods, `invokeJNINativeMethod` and `invokeJNISynchronizedNativeMethod` are the two invokers provided. In general, they must check and expand the Java stack if necessary, set up a new stack frame for the method being called, and relocate the arguments, given in the calling stack frame, according to the calling convention for the native languages on the given operating system and hardware platform. On Irix/MIPS platforms,

this means moving the arguments conditionally into integer or floating point registers or into memory in a slightly different layout.

Sun has defined a JIT API which provides the services that a JIT compiler needs from the VM. It includes pointers to these two JNI invokers as the standard way of calling JNI methods from JIT-compiled methods. However, using them introduces another preceding layer of overhead, setting up the Java stack and translating from the JIT calling convention to that expected by the invoker. Because in our JIT the calling convention and stack frame handling have been designed with the performance of calls between JIT-compiled methods as the first priority, this path of converting to the invoker interface is a little more expensive, involving setting up another stack frame and moving some of the method arguments around twice.

In all, this 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 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.

So we chose to write an optimized version of this layer, not necessarily with the goal of solving the complete problem, but of solving the cases that were easy and would at least improve the standard library performance. For Irix/MIPS, our JIT calling convention and the JNI calling convention are actually very similar in cases where all the arguments fit into the integer and floating point argument registers; the only difference is that the JNI calling convention has an extra initial argument for the JNI environment data structure. So it turned out to be true for all native methods which took up to six arguments (which a quick check showed to be true for all native methods in the standard libraries), that we could simply shift all of the arguments over within the set of argument registers and then put the JNI environment pointer in the first argument register.

For arguments of object types, things are actually slightly more complicated; the object reference passed in the JIT calling convention must actually be stored on the stack, and its address on the stack passed in the new register, since JNI passes addresses of object handles around rather than the handles themselves (the JNI type `jobject` in the JavaSoft implementation is a pointer to an object handle). We actually have two versions of our "invoker":

- One for all methods with 0–6 arguments, which contains the required logic to do exactly what is necessary for the argument and return types involved.
- One for methods with 0–6 arguments which pass/return no object types, which simply shifts all the integer and floating point argument registers unconditionally, since register moves are much cheaper than branching control structures. This version is faster for the cases where it is applicable.

In the end, 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. The benefits of this can be seen in the performance results in the following table (*Table 3*). Also note that performance gains were obtained without any changes to the MIPSJIT but by moving to Java2.

Test	SGI 116 mipsjit	SGI Java2 mipsjit	SGI Java2 mipsjit+fastJNI	fastJNI-Speedup
Native Method	98.0	69.3	21.1	3.1
Param-scalar	128.2	104.2	22.3	4.5
Param-array	126.7	74.0	31.4	2.4
Param-scalar2	140.8	110.0	23.1	4.8
Param-array-extract	410.8	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 IRIX. After looking further we found two reasons for this:

- The implementation of locking in file I/O, specifically in read(), does not scale for multi-threaded applications, as the threads must contend for certain locks.

```
JNIEXPORT void JNICALL
Java_com_lgc_prowess_dataset_Compressor_native_1uncompress16
(JNIEnv* env, jobject caller, jbyteArray jcTrace,
 jint nsmptrc, jfloatArray jtrace) {
  jboolean isCopy;
  float* trace = (float *)env->GetPrimitiveArrayCritical(jtrace, &isCopy);
  char* cTrace = (char *)env->GetPrimitiveArrayCritical(jcTrace, &isCopy);
  trcUncompress16(cTrace, nsmptrc, trace);
  env->ReleasePrimitiveArrayCritical(jcTrace, cTrace, 0);
  env->ReleasePrimitiveArrayCritical(jtrace, trace, 0);
}
```

Figure 2: **CONTENTION was created for the JNI pinning lock by multiple threads trying to use the JNI `GetPrimitiveArrayCritical` call. The underlying implementation in JNI acquires the JNI pinning lock to lock a hash table of global references when adding new references to it. By changing this to use one lock per bucket instead of one lock around the entire hash table the contention was largely avoided.**

- Contention in a lock used by the JNI code, described below.

The application's structure had multiple threads calling native code shown in *Figure 2*. Using the Speedshop performance tools and a tracing-enabled pthreads library, we found that there was one lock that was heavily contested as the number of threads increased. This was the JNI pinning lock, which was around the code which added new entries to the global reference hash table maintained by JNI. By changing to a lock-per-bucket strategy (the hash table contained 151 buckets) this contention was largely avoided and a very large performance improvement resulted, as the following table (*Table 4*) shows.

Threads	Data Throughput (single lock)	Data Throughput (Hash)	Speedup
2	10 MBytes/sec	70 MBytes/sec	7
8	6.5 MBytes/sec	35 MBytes/sec	6

For larger number of threads, the improvement was not as big but still very significant. We believe that at that point the file descriptor table lock in the kernel becomes a much bigger factor.

6 Related Work and Future Research

Matt Welsh et. al. [3] at Berkeley have studied the problem of providing very efficient communication and I/O from Java. They extend the Java runtime with new primitives that are inlined at compile time and provide external objects which are Java objects but whose storage is outside the Java heap. Chi-Chao Chang et. al. [17] looks at the problem of eliminating copies between the Java heap and the native heap, and they introduce a new buffer class and extra features in the garbage collector that eliminate the need for extra copies. Dennis Gannon et. al. [18] at Indiana University have studied the problem of object-interoperability in the context of RMI and Java/HPC++.

Our work is within a commercial product, the SGI Java2 JIT and JVM, and demonstrates that fast JNI implementations are necessary for developing multi-language Java applications. We are in the processes of developing intrinsics in the JIT compiler similar to the primitives in Jaguar for the common cases of calling into native code (e.g., `Math.sin` etc.). On many of the 64-bit Operating Systems such as IRIX and Solaris8 there is no way to call into a 64-bit native library. One of the ways of solving this is by porting the JVM to be 64-bit compliant. We have not started this work and are in the process of understanding the issues involved at this time. The Compaq JVM [19] for Digital UNIX is 64-bit and so is CACAO [20]; this provides indications that under some environments linking with 64-bit libraries is possible.

7 Conclusions

This paper discussed 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. The lessons learned from our experiments include:

- JVMs with support in the JIT for making fast JNI calls have significantly lower costs for calling a native method. But the cost of making a JNI call is still much higher than a regular Java method.
- The current support in the JVM and JNI API's is insufficient for industrial server side applications that want to embed a JVM.
- The implementation for fastJNI that we presented in the MIPS JIT and the Java VM gave substantial performance improvement over the products we shipped earlier and has improved the performance of real-world applications 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 *To appear 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 *Appears in Proceedings of the ACM 1999 Java Grande Conference*, June 1999. `http://www.cs.ucsb.edu/conferences/java99/program.html`.

[5] John Banning, Todd Smith, and Suresh Srinivas. The SGI MIPS JIT Compiler. Showcase Slides, Feb 1997.

[6] Sun Microsystems Inc. *Java Native Interface Specification*. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html`.

[7] NaturalBridge. *BulletTrainTM optimizing compiler and runtime for JVM bytecodes*. `http://www.naturalbridge.com/`.

[8] Tower Technologies. *TowerJ3.0 A New Generation Native Java Compiler And Runtime Environment*. `http://www.towerj.com`.

[9] Microsoft. *Using j/Direct to call the Win32 API from Java*. `http://www.microsoft.com/mind/0198/jdirect.htm`.

[10] Microsoft. *Integrating Windows and Java using Microsoft RNI.* http://www.microsoft.com/java/resource/techart.htm.

[11] Cygnus and Transvirtual Technologies. *Kaffe Native Interface.* http://www.cygnus.com/product/javalang/native++.html.

[12] SGI JDK 3.1.1 (based on Sun's 1.1.6). http://www.sgi.com/Products/Evaluation/#jdk_3.1.1.

[13] Java2 Software Development Kit v 1.2.1 for SGI IRIX. http://www.sgi.com/developers/devtools/languages/java2.html.

[14] Blackdown JDK port of Sun's Java Developer's Toolkit to Linux. http://www.blackdown.org/.

[15] 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.

[16] The Java Grande Forum. http://www.javagrande.org.

[17] Chi-Chao Chang and Thorsten von Eicken. Interfacing Java with the Virtual Interface Architecture. In *Appears in Proceedings of the ACM 1999 Java Grande Conference*, June 1999. http://www.cs.ucsb.edu/conferences/java99/program.html.

[18] Dennis Gannon, F.Berg, and et. al. Java RMI Performance and object model interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10:941–946, 1998.

[19] Compaq's Fast JVM for Alpha Processors. http://www.digital.com/java/alpha/index.html.

[20] Andreas Krall and Reinhard Grafl. CACAO - A 64 bit JavaVM Just-in-Time Compiler. In *Appears in Proceedings of the PPoPP'97 Workshop on Java for Science and Engineering Computation*, June 1997. http://www.complang.tuwien.ac.at/andi/articles.html.