# Implementing JNI in Java for Jalapeño

Ton A. Ngo and Steve E. Smith
*IBM T. J. Watson Research Center*
*Yorktown Heights, NY 10598*
ton@us.ibm.com, steves@us.ibm.com, http://www.research.ibm.com/jalapeno

Developed at the IBM T. J. Watson Research Center, Jalapeño [1] is a Java virtual machine (JVM) written in Java that targets high-performance servers. The strategic decision early in the project to implement Jalapeño in Java [2] leads to many important advantages, but also several implications; therefore when the Java Native Interface (JNI) was implemented for Jalapeño, we were heavily influenced by this philosophy. In this short paper, we present a JNI implementation that is written in Java[1]. We discuss the advantages and implications that arise from a Java implementation.

## 1 JNI Functions in Java

The bulk of the JNI specification [3] consists over 200 functions accessible through a JNI environment pointer that allow native code to access Java objects or invoke Java methods in the JVM. In large part, these have a similar functionality to the standard Java reflection interface. Since Jalapeño already implements Java reflection based on Jalapeño's own low-level internal reflection interface, there is a strong motivation to reuse the same internal reflection interface instead of adding a separate interface to access the JVM internal structures.

In Jalapeño it is natural that we implement the set of JNI functions in Java, rather than in C as may be expected. This allows most of the functions to implement the required functionality by simply invoking the appropriate internal Jalapeño reflection methods. Many cases only require a few lines of Java code.

[1] Except for a small set of JNI functions intended to be invoked outside the JVM, e.g. to create a JVM. These functions by necessity are written in C.

This approach leads to two important software engineering benefits. First, any internal change in Jalapeño is transparent to the JNI implementation. Second, being written in Java, the implementation for these JNI functions is completely portable when Jalapeño is ported to a new platform, even though it is by definition a native interface.

## 2 Stub compiler

For a Java method to call a native function and vice versa, we must be able to transfer control between the two environments. Since Jalapeño uses its own convention for stack frames and registers, this transfer consists of mapping between Jalapeño convention and the native convention. Since Jalapeño is a compile-only JVM, a runtime stub compiler generates the prologue and epilogue surrounding the native procedure or Java method that is being called. The special prologue establishes a transition frame on the stack to conform to the callee's convention by shuffling the parameters in the stack frame and in the registers. Similarly, the epilogue ensures that the return value matches the expected convention in the caller.

The transfer from Java to C involves all native procedures that implement Java native methods. These procedures are normally packaged in a library that is loaded from a Java program. As a class is loaded in Jalapeño, its native methods are linked to a special static method in the compiler. When a native method is invoked for the first time, this special method attempts to resolve the native method with the corresponding procedure in the library. It then invokes the stub compiler to generate the prologue and epi-

logue and links it with the native procedure found in the library. Finally, the new code is backpatched to become the actual code for the native method. A benefit with this lazy compilation approach is that only native methods that are called require compilation.

The transfer from C to Java involves the JNI functions described above. For convenience, these functions are collected in one class that implements a special *nativeInterface*. When this class is loaded and its methods are dynamically compiled into machine code, the runtime compiler recognizes the special nativeInterface and invokes the stub compiler to generate the necessary prologue and epilogue.

# 3 References and GC

For research purposes, Jalapeño hosts a family of dynamic compilers and type accurate garbage collectors; therefore it is important that no limitation on the GC policy arises from a JNI implementation. The JNI specification provides for this capability by only allowing the native code to operate on Java objects through the well-defined interface.

In Jalapeño JNI, no direct pointer to a Java object is passed to the native code since this would prevent the GC from scanning for the pointer and updating it. Instead, each Java object to be passed to the native code is saved in a side stack and an ID is given to the native code in its place. When the native code returns an object ID to the caller or passes an object ID to a JNI function (in Java), the actual object is retrieved from the side stack. The prologue and epilogue perform the pushing and popping of objects to/from the side stack as needed.

During a garbage collection cycle, the stack is scanned for references. All Jalapeño compilers implement a *StackMapIterator* class which presents a common interface for reporting the location of live references in the stack frames of the methods they have compiled. The Jalapeño JNI implementation follows this mechanism by providing a *StackMapIterator* which is invoked for each contiguous sequence of native frames in the stack. The iterator then consults the side stack to report the location of references associated with these native stack frames. This approach allows native codes to conform to a uniform garbage collection interface that accommodates all of Jalapeño garbage collectors.

# 4 Conclusions

Written in Java and interfacing directly with the internal JVM reflection, the Jalapeño JNI implementation is simple, portable, and transparent to changes in the JVM internal structures. The scheme for capturing all objects passed to the native codes allows any garbage collection policy to be explored with no limitation.

Currently, our JNI implementation on the PowerPC/AIX platform is completed and we are porting to the Intel/Linux platform. This will only require rewriting the stub compiler for the Linux stack and register convention. The few JNI functions written in C should only require some minimal porting effort. We are also investigating issues concerning the interaction between Java and native programs.

# References


[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. *The Jalapeño Virtual Machine*, IBM Systems Journal, 2000, Vol 39, No 1, pp 211-238.

[2] B. Alpern, D. Attanasio, J. Barton, A. Cocchi, S. Hummel, D. Lieber, T. Ngo, M. Mergen, J. Shepherd, and S. E. Smith, *Implementing Jalapeño in Java*, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), November 1999, pp 314-324.

[3] S. Liang, *The Java Native Interface, Programmer's Guide and Specification*, Addison-Wesley Publishers (1999).