Creating Java to Native Code Interfaces with Janet

Marian Bubak[a,b], Dawid Kurzyniec[c], Piotr Łuszczek[d,*] and Vaidy Sunderam[c]
[a]*Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland. Tel.: +48 12 617 39 64; Fax: +48 12 633 80 54*
[b]*Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland*
[c]*Emory University, Atlanta, Department of Math and Computer Science, 1784 North Decatur Road, Suite 100, Atlanta, GA 30322, USA*
Tel.: +404 712 8665
[d]*University of Tennessee, Knoxville, Department of Computer Science, 1122 Volunteer Blvd., Suite 203, Knoxville, TN 37996-3450, USA*
Tel.: +865 974 8295; Fax: +865 974 8296

Java is growing in appropriateness and usability for high performance computing. With this increasing adoption, issues relating to combining Java with existing codes in other languages become more important. The Java Native Interface (JNI) API is portable but too inconvenient to be used directly owing to its low-level API. This paper presents Janet – a highly expressive Java language extension and preprocessing tool that enables convenient integration of native code with Java programs. The Janet methodology overcomes some of the limitations of JNI and generates Java programs that execute with little or no degradation despite the flexibility and generality of the interface.

1. Introduction

In only a few years Java has evolved from a web and embedded-device programming language to a powerful general-purpose framework for numerous applications on a variety of hardware platforms. It has already penetrated the enterprise market, is gaining increasing adoption in the field of scientific computing [1,8,14,20, 22,25], and is even encountering use in system level programming and real-time systems.

There are multiple reasons for such an interest in Java technology. The "write once, run anywhere" phrase has evolved into much more than just a catchword and in most cases, Java allows applications to run unmodified on different architectures as well as in heterogeneous environments.[1] Simplicity of the language gives developers an opportunity to focus on the problem at hand, rather than syntax nuances and compiler distractions. Automated memory management helps in avoiding common programming mistakes, thus reducing debugging time. Security concerns are addressed through the ability of a Virtual Machine to restrict access to underlying operating system facilities. Yet another feature is dynamic code loading that enables Java byte code to be downloaded from a network or even to be generated on-the-fly during program execution.

The performance of modern Java Virtual Machines is already close to that of native code, and continues to improve [6,11] over time. Although this most commonly cited issue of performance is fast becoming a non-issue, there are still compelling reasons to use legacy native codes in conjunction with Java. From the software engineering perspective, reuse of existing codes with a resulting reduction in design and testing time is highly desirable. Even for code that is completely rewritten in Java, an appropriate interface to the existing version of the software makes the transition to a new implementation smoother.

The Java Native Interface (JNI) [15,16,18] defines a platform-independent API for interfacing Java with native languages such as C/C++ and Fortran. Unfortunately, its level of abstraction is rather low, which makes JNI error-prone and inconvenient to use, and results in large codes that are difficult to debug and maintain. In this paper, we present the Janet (JAva

*Corresponding author: Piotr Łuszczek, Department of Computer Science, 1122 Volunteer Blvd., Suite 203, Knoxville, TN 37996-3450, USA. Tel.: +865 974 8295; Fax: +865 974 8296; E-mail: luszczek@cs.utk.edu.

[1]The GUI-related Java portability issues rarely apply to high-performance applications.

Scientific Programming 9 (2001) 39–50
ISSN 1058-9244 / $8.00

Native ExTensions) project – a language extension that provides a preprocessing tool and enables convenient development of native methods and Java interfaces to legacy codes. Janet facilitates the use of JNI so that no explicit calls to the JNI API have to be made. Further, Janet allows Java and native codes to coexist in the same source file which contributes significantly to clarity and readability.

The remainder of this paper is organized as follows: Section 2 discusses similar projects, while Section 3 provides an overview of JNI. Sections 4, 5, 6, and 7 describe the Janet tool in detail including syntax and semantics aspects. In Section 8 performance results for Janet interfaces are presented, including benchmarks for wrappers to a parallel library called LIP. Finally, conclusions and future work are discussed in Section 9.

2. Related work

The JCI tool is an automatic Java interface generator for C language codes [7,21]. As input, the tool accepts a C header file with declarations of native library functions, and generates JNI wrappers for these functions. Although such an automated approach is very convenient, the lack of provisions for user input, combined with substantial semantic differences between Java and C unavoidably leads to wrappers that do not conform to the Java programming style, i.e., they are not object oriented, unsafe in many respects, and use function return codes rather than exceptions to report erroneous situations.

The Jaguar project [24] introduces extensions to the JIT technology and Java bytecode. It bypasses the JNI layer and enables direct access to the underlying computing platform. This approach is promising as it leads to much more efficient codes than those employing JNI. Unfortunately, it also binds the resulting code to a particular architecture (currently only Intel x86) and is therefore not very flexible. In contrast, our approach is solely based on JNI and thus retains a high level of portability.

The Jalapeño project [12] implements a Java Virtual Machine written almost entirely in Java itself. Due to the fact that a VM must be able to access main memory directly without the usual safety restrictions, the authors use a special MAGIC class with methods implemented in machine code. The Jalapeño project emphasizes high performance rather than portability and as such is not intended to provide general purpose support for interfacing Java with native languages.

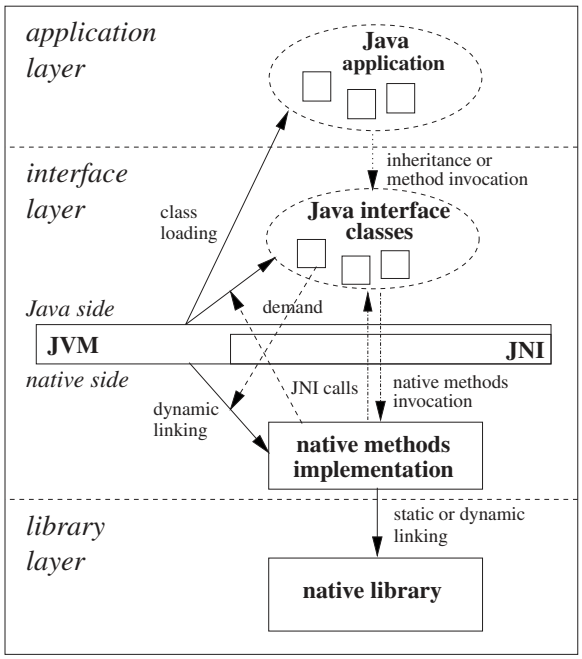

Fig. 1. Use of a native library in a Java application.

An alternative way of wrapping legacy code is to use *shared stubs* [18]. This technique allows invocation of arbitrary functions residing in shared libraries and uses system-level routines for dynamic linking. Such a technique is, again, platform-dependent and introduces substantial overheads in native function calls. Nonetheless, Janet can be used simultaneously with this approach as they do not exclude each other.

3. Java Native Interface overview

JNI [15,18] is an Application Programming Interface (API) that allows Java code (running inside a Java Virtual Machine) to interoperate with applications and libraries written in other programming languages, such as C/C++ or Fortran. One of the most important benefits of JNI is that it imposes no restrictions on the implementation of the underlying Java VM.

JNI allows the implementation of Java methods which have been declared as `native` in a class definition. Figure 1 shows interactions between a Java application, the Java VM, JNI and native code in the situation where native methods serve as an interface between the Java application and the legacy native library.

The essential feature of JNI is that it allows native code to have the same functionality as pure Java code. In particular, it provides means to create, inspect

```
(a) array access

JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr) {
    jint *carr; jint i, sum = 0;
    jint len = (*env)->GetArrayLength(env, arr);
    if (!(carr = (*env)->GetIntArrayElements(env, arr, NULL))) return 0;
    for (i=0; i<len; i++) sum += carr[i];
    (*env)->ReleaseIntArrayElements(env, arr, carr, 0);
    return sum; }

(b) field access

JNIEXPORT void JNICALL
Java_lip_Lip_maptableFree(JNIEnv *env, jclass cls, Maptable mtab) {
    jclass mtc; jfieldID fid; jlong l;
    if (!(mtc = (*env)->FindClass(env, "lip/Maptable"))) return;
    if (!(fid = (*env)->GetFieldID(env, mtc, "data", "J"))) return;
    l = (*env)->GetLongField(env, mtab, fid);
    LIP_Maptable_free(l); }

(c) method invocation and exception handling

JNIEXPORT void JNICALL
Java_dummy_method(JNIEnv *env, jobject obj) {
    jthrowable exc;
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "()V");
    if (mid == NULL) return;
    (*env)->CallVoidMethod(env, obj, mid);
    if (exc = (*env)->ExceptionOccurred(env)) {
        jclass newExcCls;
        JNI_EXCEPTION_DESCRIBE();
        if (!(newExcCls = (*env)->FindClass(env,
                    "java/lang/IllegalArgumentException"))) return;
        (*env)->ThrowNew(env, newExcCls, "from C code"); }}

(d) synchronization

JNIEXPORT void JNICALL
Java_dummy_foo(JNIEnv *env, jobject obj, jobject bar) {
    (*env)->MonitorEnter(env, bar);
    native_foo();
    (*env)->MonitorRelease(env, bar); }
```

Fig. 2. Examples of native methods using JNI features.

and modify objects (including arrays), invoke methods, throw and catch exceptions, synchronize on Java monitors and perform runtime type checking by calling appropriate functions of the JNI API. Several examples are shown in Fig. 2.

The main problem in using JNI is the fact that it is much closer to the Java VM than Java language itself, so its level of abstraction is rather low. This makes the development process long, inconvenient and error-prone. Most of the programming mistakes (which are rather easy to make) in the created interfaces lead to undefined behavior at runtime, resulting in hard-to-track and platform-dependent errors. The following are examples of such error-prone situations.

- In order to access Java arrays of primitive data types, native code must invoke a special JNI function to lock an array and obtain a pointer to it. When the array is no longer being accessed, another function must be called to release the array. The JNI specification does not define the behavior of a program which fails to release an array. In addition, access functions to arrays of different types have different names. The use of improper functions causes runtime errors rather than compilation warnings (see Fig. 2(a)).
- Accesses to fields of Java objects must be performed through opaque field descriptors. To obtain such a descriptor, the user must know not only the name of the field, but also a *signature* string of the field type. Again, native methods compile with no errors but fail at runtime if the type of the accessed field has changed. Moreover, JNI provides separate functions for each type and storage attribute of a given field (see Fig. 2(b)).
- Invocation of Java methods is even more complicated. Methods are invoked through opaque descriptors that are obtained for a specific method name and signature. However, the method sig-

nature depends on the types of all its parameters, which makes native code even more sensitive to changes in Java classes on which it depends. As before, distinct JNI functions are required for invocation of methods with varying return types and *invocation modes*, i.e., instance, static, and non-virtual (see Fig. 2(c)).
- Exceptions that may occur in native methods, e.g., as a result of a JNI call that invokes a Java method, cannot be handled like an ordinary Java exception and caught by a Java exception handler. Instead, an explicit query is necessary. This query is mandatory since the behavior of subsequent JNI calls is undefined when there are pending exceptions (see Fig. 2(c)).
- Lock and unlock operations on Java monitors are independent of each other, so frequently the latter is mistakenly omitted at runtime, especially within exception handling code (see Fig. 2(d)).

4. Overview of Janet

Janet is a Java language extension and preprocessing tool that enables the convenient development of native methods and Java interfaces to native code by removing the need for explicit calls to JNI.

With JNI, the definitions of native methods must be written in separate source files. In contrast, Janet allows Java and native source codes to coexist in a single file.[2] Moreover, such an embedded native code can directly perform Java operations such as accessing Java fields and variables, invocation of Java methods, use of Java monitors, exception handling, etc. These operations can be carried out using ordinary Java language syntax – no cumbersome JNI function calls are required.

Currently, the only native language that is supported is C. However, due to the open architecture of Janet, support for other languages may be added with little effort.

A Janet file is transformed by Janet preprocessor into Java and native language source files, as shown in Fig. 3. The translation process separates a native code from Java code, and inserts appropriate JNI function invocations. The JNI code, automatically generated for the user, performs the following operations: it determines necessary type signatures, chooses JNI functions

[2] A native code can appear as an implementation of a `native` method, or inside newly introduced `native` statements analogous in syntax to the `synchronized` statement.

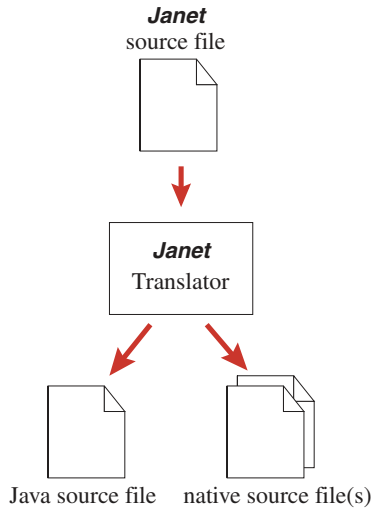

Fig. 3. Janet translation process.

to call, loads Java classes, obtains field and method descriptors, performs array and string lock and release operations, handles and propagates Java exceptions, and matches monitor operations.

A simple example of the canonical "Hello World" program that uses Janet (file `HelloWorld.janet`), is presented below. This example demonstrates the use of a single native method whose body is embedded in Java source code.

```
class HelloWorld {
native "C" {
#include <stdio.h>
}
  public native "C" void display
  HelloWorld() {
    printf("Hello world!\ n");
  }
    public static void main
    (String[] args) {
  }
}
```

The translation process generates three source files in this case: the first contains stripped Java source with only a native method declaration, the second contains the native method implementation, and the third is an auxiliary C source file. The first two files are presented below:

- File `HelloWorld.java`:

```
class HelloWorld {
  /* ... code that loads library
  goes here ... */
```

```
  public native void displayHello
  World();
  public static void main(String
  args) {
  new HelloWorld().displayHello
  World();
  }
}
```

– File `HelloWorldImpl.c`:

```
#include <janet.h>
#include <stdio.h>
Janet_HelloWorld_displayHello
World(JNIEnv *_janet_jnienv,
      jobject _janet_obj)
{
  printf(``Hello world!\ n'');
}
```

Next, all native source files must be compiled into a shared library `HelloWorld` (e.g., `libHelloWorld.so` on Linux and Solaris operating systems).

A Java VM will search for `libHelloWorld.so` during the initialization of the class `HelloWorld` (for brevity, the code for linking the library is not shown). The name for a library is established by the Janet preprocessor using the following naming convention – if the class appears in the default (unnamed) package as it does in this example, the class name is used as the library name. Otherwise, the package name (*mangled* if necessary, see [15,18]) is used instead. In future releases, we will add more flexibility in assigning names to native libraries.

The following two sections present several examples of Java operations (expressions and statements) that are allowed to appear inside native code (embedded within back-tick characters, `` `...' ``) and that are transformed by Janet preprocessor into JNI function invocations. Up-to-date source code for all the examples as well as detailed documentation can be obtained from [13].

5. Embedded Java expressions

5.1. Simple expressions

Field access operations belong to the most commonly performed JNI operations. Janet simplifies these operations by allowing Java syntax to be embedded directly inside the C code. The following example is taken from the interface to the LIP [5] library:

```
static native void maptableFree
(Maptable mtab) {
  LIP_Maptable_free(`mtab.data');
}
```

In this example, the C function `LIP_Maptable_free` receives a parameter whose value is fetched from the `data` field of an object of `Maptable` class which in turn is passed as a parameter to the Java method `maptableFree`.

Other Java expressions that are slightly more complicated but also commonly used inside native methods are method invocations. Consider the following example:

```
class C {
  int bar(int a, int b) { ... }
  int bar1() { throw new Runtime
  Exception(); }
  int bar2() { ... }
  native void foo() {
    some_C_routine(`bar(bar1(),
    bar2())');
  }
}
```

Native method `foo` contains a Java expression (as a parameter of `some_C_routine`) that contains three invocations of Java methods. Janet translates these pieces of code into an appropriate sequence of JNI calls. What is important in this example is the preservation of exact Java semantics (see [9] §15.5 and §15.6), e.g., in the case when the method `bar1` throws an exception, neither the `bar` nor `bar2` methods are invoked. In general, the native code generated by Janet guarantees the same evaluation order and precise exception handling as Java code does.

5.2. Native subexpressions of embedded Java expressions

Let us consider a situation when the user wants to invoke a Java method `bar` that has to be passed the value of some native variable as a parameter. Since Java and native code namespaces are separated by Janet, such an operation requires the user to specify explicitly that a given expression references native code's namespace. This is done with a #(*expr*) syntax as in the following example:

```
native void foo() {
  int i = 0;
  `bar(i, 0)':  /* compilation
  error:  what ss `i'?  */
```

```
  'bar(#(i), 0)'; /* OK: 'i' comes
  from native code side */
}
```

The Java type of such an embedded subexpression is inferred from the context (the original native type is not considered). In the example, Janet casts the expression to this type before passing the value from the native method because the first parameter of method `bar` has Java type `int`. In ambiguous cases, the type cast must be performed explicitly:

```
class C {
  void bar(int i) { ... }
  void bar(boolean b) { ... }
  native void foo() {
    int i = 0;
    'bar(#(i))'; /* compile error:
    ambiguous */
    'bar((boolean)#(i))'; /* OK */
  }
}
```

Embedded native expressions are not limited to simple variable accesses as in the above examples. In fact, they may be fairly complex and may even contain embedded Java expressions. This process of embedding may be repeated recursively.

5.3. Accessing arrays

Arrays are simple and yet powerful data structures. They are probably the most commonly used in scientific codes. A large number of native libraries operate on arrays and they could be effectively used in Java if an appropriate interface is provided. For this reason, we considered it crucial for Janet to provide efficient and convenient support for operations on Java arrays.

Janet allows embedding Java array access expressions into native code in the same fashion as other expressions. Consider the following code example of a native method which computes the sum over elements of a Java array:

```
native int sum(int[] arr) {
   int i, sum = 0;
   for (i = 0; i < 'arr.length';
   i++) {
     sum + = 'arr[#(i)]'; /* C
     variable indexes Java array */
   }
   'return sum;'
}
```

In this example we have two embedded Java expressions: one is the read of the array field `length`, and the other performs array access. Note that the array is indexed using the native variable `i`, and there is no type ambiguity as the array index in Java has always type `int`. Janet takes advantage here of the fact that the field `length` is `final`. Thus, the JNI routine that obtains `length`'s value is invoked only once, even though the expression that uses it is evaluated multiple times. Additionally, the array access itself is optimized: Janet obtains the pointer to the whole array during the first access, and once it is done, subsequent iterations are executed without any JNI calls. Finally, the array pointer is released at the end of the method `sum`.

Such an array access scenario, although very common, is still not sufficient. If, for example, one wants to use a legacy native routine which operates on the array passed as a pointer argument, the pointer to the Java array must be obtained explicitly. To solve this problem, Janet introduces address-fetch operator & which can be used with arrays and strings.[3] As an example, consider how the sorting routine `qsort` from the standard C library can be used in a Java method `qsort` (note again separate namespaces for C and Java):

```
native void qsort(int[] arr) {
   jint* ptr;
   ptr = '&arr';
   qsort(ptr, 'arr.length',
   sizeof(jint), ...);
}
```

The last concern is that Java arrays store platform-independent primitive data types rather than native ones, and these are not necessarily the same.[4] The & operator does not perform any type conversion – it simply exposes the array as it is. If explicit conversion is desired, the `#&` operator may be used:

```
native void polint(float[] xa,
float[] ya, ...) {
  /* assume that polint() accepts
  native C float[] */
  polint('#&xa', '#&ya', ...);
}
```

[3] Due to JNI limitations, the address-fetch operator & cannot be used with arrays of reference types, including multidimensional arrays.

[4] JNI defines `jint`, `jlong`, `jboolean`, `jchar`, `jbyte`, `jshort`, `jfloat` and `jdouble` as native equivalents of Java primitive data types.

Table 1
Java to C type mapping for & and #& operators

Java type	& operator	#& operator
boolean[]	jboolean*	unsigned char*
byte[]	jbyte*	signed char*
char[]	jchar*	unsigned short*
short[]	jshort*	short*
int[]	jint*	int*
long[]	jlong*	long*
float[]	jfloat*	float*
double[]	jdouble*	double*
String (UNICODE)	const char* (UNICODE)	const char* (UTF-8)

The #& operator can also be applied to Java strings, converting them from UNICODE to UTF-8 format:

```
native void print(String s) {
  /* simple Java strings can be
  printed from C */
  printf('#&s');
}
```

The mapping of types from Java to C that is performed by & and #& operators is shown in Table 1. The array conversion performed by the #& operator introduces no performance degradation on platforms where appropriate array element types are equivalent. However, it requires allocation and copying of the entire array in cases when they are different.

6. Embedded Java statements

6.1. Declaring variables

Java variables can be declared inside a native method implementation and used in subsequent embedded Java expressions:

```
native void method(BookStore bs) {
   'Book b;'
   ...
   'b = bs.getBook();'
   ...
   printf(''%d\n'',
   (int)'b.getPageCount()');
}
```

Additionally, such variables give explicit control over occurrences of the array get/release operations (Fig. 2) in the code generated by Janet. Arrays are not released as long as any variables referencing them remain in the current scope. Otherwise, they are released upon reaching the end of the block that surrounds the array access expression, or when a different array reference is produced by an expression. This is shown in the following sample code:

```
class Dummy {
  int[] arr0, arr1;
  ...
  native void foo() {
    'int local[];'
    {
      'arr0[0]'; /* get contents
      of arr0 */
      'local = arr1;' /* new
      reference to arr1 */
      'local[0]'; /* get contents
      of arr1 */
      } /* arr0 released (end of
      block) */
    ...
  } /* arr1 released ('local' goes
      out of scope) */
}
```

6.2. Exception handling

Exception handling is one of the most error-prone aspects of JNI. The user must explicitly check for exceptions in every possible place where they may occur (essentially after every JNI call) and provide code to handle them. As exceptions usually break normal flow of program execution, it becomes easy to mismatch array or monitor lock/release operations within exception handling code. In contrast, Janet provides a convenient syntax for exception handling, by adapting Java's exception model and employing `try`, `catch`, `finally` and `throw` statements. The following example illustrates these concepts:

```
native void method() {
  'try {
    callback();
  } catch (Throwable e) {
```

```
    'JNI_EXCEPTION_DESCRIBE();'
    throw new IllegalArgument
    Exception(''from C code'');
  }'
}
```

Again, the semantics of the generated native code strictly conforms to the Java language definition. In particular, exceptions are handled as soon as they occur, arrays and monitors are guaranteed to be always released, and the `finally` clauses are always evaluated. The only segment of native code involved is the `JNI_EXCEPTION_DESCRIBE` macro call – the remainder is Java code that handles exceptions. Such a syntax simplification is possible because Janet allows the merging of subsequent embedded Java operations, which eliminates extra back-tick delimiters. (Compare this example with the analogous JNI code shown in Fig. 2(c).)

6.3. Synchronization

JNI provides separate functions for monitor lock and unlock operations. In contrast, Janet adapts the Java `synchronized` statement for this purpose:

```
native void foo(Object bar) {
  'synchronized(bar)' {
    native_foo();
    }
}
```

Again, the generated code is exception-aware. The monitor is unlocked even if exceptions occur inside `synchronized` body. This is achieved through a construct similar to the `try` statement with a `finally` clause that contains code to unlock the monitor.

6.4. *Java-style* `return` *statement*

In general, the C language `return` statement should *not* be used inside native methods when using Janet. Instead, the Java-style `return` statement is introduced:

```
native int foo() { /* will always
return 1 */
  'try' {
    'return 0;'
  } 'finally' {
    'return 1;'
  }
}
```

There are two reasons why C's `return` statement should not be used. First, during compilation, it prevents type checking for the returned value and can potentially lead to runtime errors. Secondly, it prevents Janet from executing `finally` clauses as was required in the example above.

6.5. Unconditional branch statements

Currently Janet uses a complete parser for Java code and a simplified one for embedded C. This approach has the advantage of extensibility as it is easy to add new parsing modules for additional native languages. Also, it increases the portability of the tool and generated interfaces. However, it limits the syntax of native codes. Consider the following example:

```
do {
  'synchronized(foo)' {
    break;
    } /* monitor unlock would occur
    here */
  } while (false);
```

The Janet preprocessor does not recognize semantics of the `break` statement and inserts a monitor unlock code at the end of the block. Therefore, this code unlocks the Java monitor when the native method returns (rather than when `do-while` loop terminates). To avoid such situations, Janet forbids unconditional branch statements, namely `break`, `continue`, `goto`, as well as the `longjmp()` function call, to be used in native code if they could prematurely exit the block in which they appear. Also, the use of the `return` statement is strongly discouraged for the reasons described in Section 6.4. This issue pertains only to the C language. With C++, it is possible to avoid this problem by using object destructors.

7. Portability

One of the main goals of the Janet project is to retain a high level of portability of both the tool itself and the code it generates. The Janet preprocessor is therefore written entirely in Java and it can run on any Java 2 platform. The whole system consists of approximately 130 source files and 30,000 lines of code. The generated C source code fully conforms to the ANSI C standard and may be used with JNI version 1.1 onwards; therefore, it works with JRE 1.1. At the same time, it can also take advantage of the JNI 1.2 extensions introduced in Java 2.

Table 2
Performance results on Solaris OS with JDK 1.3 and HotSpot VM (time shown in microseconds)

	Java	JNI	Janet
private native method inv.	0.11 + 0.025*argc		
virtual native method inv.	0.14 + 0.025*argc		
private method invocation	0.04 + 0.005*argc	6.7 + 0.9*argc	7 + 0.9*argc
virtual method invocation	0.05 + 0.005*argc	12.5 + 0.9*argc	13 + 0.9*argc
field access	0.025	0.45	0.5
dynamic cast	0.035	0.4	0.45
`try` (no exception)	0.045	0	0.25
`throw`	20	50	50
`catch` (exception thrown)	0	1	1.5
`synchronized`	0.2	1.5	1.5
array access (normal)		8*size	8*size
array access (fast)		0.75	20
per-method – arrays & locks			1.5
per-method – exceptions			0-0.25

argc – number of parameters passed to a method
size – size of array in KB

8. Performance results

JNI provides a highly portable and abstract interface layer, e.g., it makes no restrictions as to how a Java VM represents objects internally. While this approach facilitates writing portable native methods, it also introduces an overhead much higher than if the objects could be accessed directly. Since Janet is built on top of JNI, Janet performance is highly influenced by the performance of JNI itself. To measure these overheads, we performed a series of benchmark experiments on different platforms.

Table 2 shows performance results for the HotSpot Java VM from Java 2 Standard Edition v1.3 for Solaris. The host platform was a 4-processor Sun Enterprise 450 with 4 UltraSPARC 400 MHz CPUs with 4 MB of ECache and 1280 MB RAM running SunOS 5.7. Tables 3 and 4 show performance results for HotSpot and Classic Java VMs, respectively, from the Java 2 Standard Edition v1.3 for Linux. The host platform was a PC with a Pentium II 440 MHz CPU and 128 MB RAM running RedHat Linux 6.2. All numbers show CPU time in microseconds (μs).

The test methodology was as follows. For each test, two separate functions were provided. They differed only in the use of the operation to be measured. A single test run involved a number of iterative executions of both methods, so that cumulative execution times could be compared. The number of iterations was chosen empirically (from the range of 10^3 to 10^8) for each test, to ensure low deviation between execution times and provide accuracy of at least 1.5 significant digits. Immediately before measurements were started, each Virtual Machine was allowed to execute the same number of "warm-up" iterations in order to optimize the code. The numbers in all tables are average times over at least 8 test runs. For the JNI test routines, safety features were omitted to obtain the highest possible performance, e.g., the exception checks after method invocations were not included.

To begin with, the efficiency of both private and virtual native method calls was measured as these are the basis of any native code interface. Next, a series of tests was performed to compare execution overhead of different kinds of Java expressions and statements as they appear in pure Java, in native methods written using pure JNI, and in native methods written using Janet. Then, the performance of Java array access from within native code was measured for JNI and Janet, using both traditional `Get<type>ArrayContents` JNI routines (*normal*) as well as the `GetPrimitiveArrayCritical` routine (*fast*) introduced in JNI 1.2.

Finally, the additional Janet-specific method invocation overhead was measured in the situations where arrays of primitive type or `synchronized` statements are used, and when the method handles Java exceptions, i.e., when callback method invocations are involved.

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 which is acceptable in most cases, as JNI functions typically take only a small part of the total native method execution time. Therefore, overall JNI performance seems to be adequate for most applications. However, there are several issues that one has to be aware of:

Table 3
Performance results on Linux OS with JDK 1.3 and HotSpot VM (time shown in microseconds)

	Java	JNI	Janet
private native method inv.	0.1 + 0.012*argc		
virtual native method inv.	0.12 + 0.012*argc		
private method invocation	0.02 + 0.002*argc	4.2 + 0.65*argc	5 + 0.65*argc
virtual method invocation	0.022 + 0.0035*argc	9 + 0.65*argc	10 + 0.65*argc
field access	0.005	0.28	0.3
dynamic cast	0.02	0.16	0.18
`try` (no exception)	0.002	0	0.45
`throw`	36	74	83
`catch` (exc. thrown)	0.4	1–2.5	1–2.5
`synchronized`	0.04	1.2	1.5
array access (normal)		21*size	21*size
array access (fast)		0.35	17
per-method – arrays & locks			1
per-method – exceptions			0–0.45

argc – number of parameters passed to a method
size – size of array in KB

Table 4
Performance results on Linux OS with JDK 1.3 and Classic VM (time shown in microseconds)

	Java	JNI	Janet
private native method inv.	0.55 + 0.05*argc		
virtual native method inv.	0.45 + 0.06*argc		
private method invocation	0.27 + 0.016*argc	1.3 + 0.06*argc	1.4 + 0.06*argc
virtual method invocation	0.27 + 0.02*argc	1.3 + 0.06*argc	1.45 + 0.06*argc
field access	0.05	0.18	0.22
dynamic cast	0.1	0.25	0.25
`try` (no exception)	0.02	0	0.45
`throw`	8	23	15
`catch` (exc. thrown)	0.28	0.5	0.5
`synchronized`	0.8	0.6	0.8
array access (normal)		1.4	6.2
array access (fast)		1.4	6.7
per-method – arrays & locks			1.5
per-method – exceptions			0–0.45

argc – number of parameters passed to a method

– Copying arrays and strings instead of pinning them down can degrade the performance substantially. Unfortunately, even `Get ... Critical()` routines (which introduce restrictions on the enclosed native code and therefore cannot be always used) 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 computation, the additional invocation overhead can exceed any native code performance benefits.
– Excessive callbacks from native methods can be expensive 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 is possible that a new VM version from the same vendor executes JNI calls less efficiently than an older version. This was the case with HotSpot VM for Linux, where the JNI implementation is much less efficient than that of the Classic VM.

A large overhead is also introduced when a native method throws an exception, but it is not a real issue because in properly written programs exceptions are thrown rarely. It was observed that the execution of the `throw` statement in pure Java code also takes an enormous amount of time.

In most cases, Janet adds no more than 20% to the JNI overhead. The additional time is spent in retaining the safety of the running code i.e., for method invocations, Janet checks if they did or did not cause an exception (note that every Java method not declared to throw exceptions may still throw `RuntimeExceptions` and `Errors`). A notable difference between JNI and Janet performance is evident only for array accesses. This

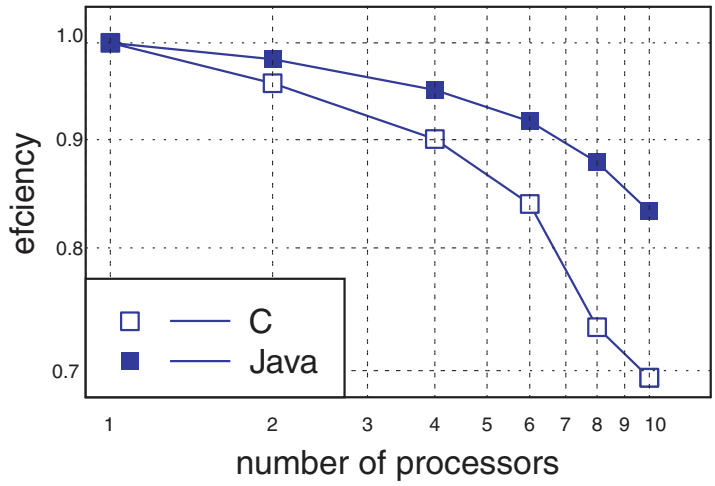

Fig. 4. Performance results of the code using the lip library from C and Java ($\mathtt{n} = 100$).

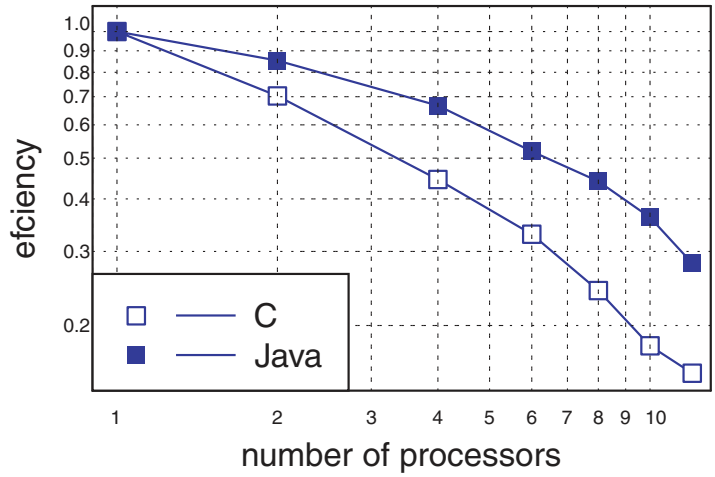

Fig. 5. Performance results of the code using the lip library from C and Java ($\mathtt{n} = 1000$).

is because Janet invokes an additional method to avoid aliasing problems with multiple references pointing to the same array. This initial overhead however, is usually amortized over the time of actual array processing and has little overall effect.

The Janet project was originally developed as a Java interface to the LIP programming library [3–5]. The LIP library is built on top of MPI [8,17,19] and supports both in- and out-of-core (OOC) parallel irregular problems [2,23] (i.e. problems that indirectly access large data arrays). To test the performance of Janet, a generic irregular OOC problem was written in Java. Its scalability in comparison to the C version is presented in Figs 4 and 5. All computations in the Java test code were performed on the Java side, while the native libraries were provided only as a communication layer and the OOC I/O environment. The amount of computation was proportional to the variable `n`, while the communication overhead remained constant across all tests. Unlike the previous results, here the performance of Janet is shown in relative terms. They are presented to give a perspective on a real-world parallel application behavior, which involves complex interactions between software and hardware components. These interactions are not present in the benchmark tests. Absolute performance values are as indicated by the previously shown test results.

These results demonstrate that Java can be efficiently employed in large scale scientific parallel computations, adding rapid software development and safety to the power of existing native computing environments.

9. Conclusions and future work

This paper has described a new approach to creating Java interfaces to native codes. The proposed Java language extensions and the Janet preprocessing tool enable simple, fast and convenient development of efficient interfaces while retaining full control over their low-level behavior. For the immediate future, our goal is to provide a visual environment (with a graphical user interface) to enable the user to graphically design the structure of Java wrappers for a native library. The tool would then generate Janet code which could be further refined by the user. A fully automatic wrapper generator is also under consideration with its output being subject to potential refinement with the GUI tool. We also intend to apply Janet to enable the usage of native resources in the Harness Metacomputing Framework [10,20]. Finally, support for native languages other than C is under development. The first candidate here is C++ as it would eliminate the aforementioned problems with unconditional branch statements.

Acknowledgments

This research was done in the framework of the Polish-Austrian collaboration and it was supported in part by the KBN grant 8 T11C 006 15.

References

[1] 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, Feb 1998, Available at http://www.cs.ucsb.edu/conferences/java98/papers/jnt.ps.

[2] P. Brezany, Input/Output Intensively Parallel Computing, *Lecture Notes in Computer Science* **1220** (1997), Springer, Berlin Heidelberg New York.

[3] M. Bubak, D. Kurzyniec and P. Łuszczek, Creating Java to Native Code Interfaces with Janet Extension, in: *Proceedings of the First Worldwide SGI Users' Conference,* M. Bubak, J. Mościński and M. Noga, ACC-CYFRONET, Cracow, Poland, October 11–14, 2000, pp. 283–294.

[4] M. Bubak, D. Kurzyniec and P. Łuszczek, A Versatile Support for Binding Native Code to Java, in: *Proceedings of the HPCN Conference,* M. Bubak, H. Afsarmanesh, R. Williams and B. Hertzberger, eds, Amsterdam, May 2000, pp. 373–384.

[5] M. Bubak and P. Łuszczek, Towards Portable Runtime Support for Irregular and Out-of-Core Computations, in: *Proceedings of 6th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science,* J. Dongarra, E. Luque and T. Margalef, eds, Springer, Barcelona, Spain, September 26–29, 1999, pp. 59–66.

[6] Osvaldo Pinali Doederlein, The Java Performance Report, http://www.javalobby.org/fr/html/frm/javalobby/features/jpr/part3.html.

[7] V. Getov, S. Flynn-Hummel and S. Mintchev, High-Performance Parallel Programming in Java: Exploiting Native Libraries, in: *ACM-1998 Workshop on Java for High-Performance Network Computing,* Stanford University, Palo Alto, California, Feb 1998, Available at http://www.cs.ucsb.edu/conferences/java98/papers/hpjavampi.ps.

[8] 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.

[9] J. Gosling, B. Joy, G. Steele and G. Bracha, *The Java Language Specification,* (Second ed.), Addison-Wesley, 2000, Available at http://java.sun.com/docs/books/jls/.

[10] Harness Project Home Page, http://www.mathcs.emory.edu/harness/.

[11] Java HotSpot Technology, http://java.sun.com/products/hotspot/.

[12] Jalapeño Project Home Page, http://www.research.ibm.com/jalapeno/.

[13] JANET Project Home Page, http://www.icsr.agh.edu.pl/janet/.

[14] Java Grande Forum, http://www.javagrande.org/.

[15] Java Native Interface, http://java.sun.com/j2se/1.3/docs/guide/jni/.

[16] Trail: Java Native Interface, http://java.sun.com/docs/books/tutorial/native1.1/.

[17] LAM/MPI Parallel Computing, http://www.mpi.nd.edu/lam/.

[18] S. Liang, *The Java Native Interface: Programmer's Guide and Specification,* Addison-Wesley, 1999.

[19] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface,* July 18, 1997, Available at http://www.mpi-forum.org/docs/mpi-20.ps.

[20] 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.

[21] S. Mintchev and V. Getov, Towards Portable Message Passing in Java: Binding MPI, in: *Lecture Notes in Computer Science,* (Vol. 1332), Springer-Verlag, Berlin-Heidelberg, 1997, pp. 135–142.

[22] M. Philippsen, Is Java Ready for Computational Science? in: *Proceedings of the 2nd European Parallel and Distributed Systems Conference for Scientific Computing,* Vienna, July 1998, Available at http://math.nist.gov/javanumerics/.

[23] J. Saltz, *A Manual for the CHAOS Runtime Library, UMIACS Technical Reports CS-TR-3437 and UMIACS-TR-95-34,* University of Maryland, March 1995, Available at ftp://ftp.cs.umd.edu/pub/hpsl/chaos_distribution/.

[24] M. Welsh and D. Culler, Jaguar: Enabling Efficient Communication and I/O in Java, *Concurrency: Practice and Experience* **12** (Dec. 1999), 519–538, Special Issue on Java for High-Performance Applications, Available at http://www.cs.berkeley.edu/~mdw/papers/jaguar-journal.ps.gz.

[25] G. Zhang, B. Carpenter, G. Fox, X. Li and Y. Wen, *The HPspmd Model and its Java Binding,* (Vol. 2), Programming and Applications, (Chapter 14), Prentice Hall, Inc., 1999.