Creating Java to Native Code Interfaces with **Janet** Extension

Marian Bubak[1,2], Dawid Kurzyniec[3], Piotr Łuszczek[4], and Vaidy Sunderam[3]

[1] Institute of Computer Science
AGH, al. Mickiewicza 30
30-059 Kraków
Poland
phone: (+48 12) 617 39 64, fax: (+48 12) 633 80 54

[2] Academic Computer Centre – CYFRONET
Nawojki 11
30-950 Kraków
Poland

[3] Emory University, Atlanta
Department of Mathematics and Computer Science
1784 North Decatur Road
Atlanta, GA 30322
U.S.A.
phone: (+404) 712 8665

[4] University of Tennessee, Knoxville
Department of Computer Science
1122 Volunteer Blvd., Suite 203
Knoxville, TN 37996-3450
U.S.A.
phone: (+865) 974 8295, fax: (+865) 974 8296

Address for correspondence:
Piotr Luszczek
Department of Computer Science
1122 Volunteer Blvd., Suite 203
Knoxville, TN 37996-3450
U.S.A.
phone: (+865) 974 8295, fax: (+865) 974 8296
e-mail: `luszczek@cs.utk.edu`

Abstract

As Java becomes an appropriate environment for high performance computing, the interest arises in combining it with existing code written in other languages. Portable Java interfaces to native code may be developed using the Java Native Interface (JNI). However, as a low-level API it is rather inconvenient to be used directly, thus the higher level tools and techniques are desired. We present **Janet** – a highly expressive Java language extension and preprocessing tool that enables convenient integration of native code with Java programs.

1 Introduction

In only a few years Java has evolved from a programming language for embedded consumer electronics to a powerful general-purpose technology being used to solve variety of problems across numerous hardware platforms. It has already penetrated the enterprise market, is gaining increasing adaptation in the field of scientific computing [25, 8, 14, 22, 1, 20], and is even beginning to cope with system level programming and real-time systems.

There are multiple reasons for such an interest in Java technology. The "write once, run anywhere" phrase evolved to much more than just a catchword and, to many, Java allows applications to run on different architectures as well as in heterogeneous environments. [1] Simplicity of the language gives developers an opportunity to focus on a problem itself rather than be hindered by syntax nuances. Automated memory management helps avoiding common programming mistakes, thus reducing the debugging time. Security concerns are addressed with the ability of a Virtual Machine to restrict an access to underlying operating system facilities. Yet another feature is dynamic code loading what 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 a native code and still keeps improving [11, 6]. As this most commonly quoted bottleneck is being quickly removed, there are still compelling reasons to use the legacy native code with Java. From the software engineering perspective, reuse of both: the design and testing time is highly desirable. Even for the code that is meant to be completely rewritten in Java, a pertinent interface to the existing software makes the transition to a new implementation smoother.

The Java Native Interface (JNI) [15, 18, 16] defines platform-independent API for interfacing Java with native languages such as C/C++ and Fortran. Unfortunately, its level of abstraction is rather low what makes JNI error-prone and inconvenient to use and results in large amounts of code that is difficult to debug and maintain. In this paper, we present the **Janet** (**JA**va **N**ative **E**x**T**ensions) project, which is a language extension that provides the preprocessing tool and enables convenient development of native methods and Java

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

interfaces to legacy codes. **Janet** facilitates the use of JNI so that no explicit calls to JNI API have to be made. Also, it allows for Java and native codes to coexist in the same source file which contributes to unprecedented clarity of expression.

The rest of this paper is organized as follows: section 2 discusses similar projects, section 3 provides an overview of JNI, sections 4, 5, 6, and 7 describe the **Janet** tool in detail – its syntax and semantics. In section 8 the performance results for **Janet** interfaces are shown, including benchmarks for the wrappers to the parallel library called `lip`. Finally, conclusions and future work are given in section 9.

2 Related Work

The JCI tool is an automatic Java interface generator for the C language codes [21, 7]. As an input, the tool accepts a C header file with declarations of native library functions, as a result, it generates JNI wrappers for these functions. Although such automatic approach is very convenient, the lack of human influence combined with substantial semantic differences between Java and C languages unavoidably leads to wrappers that do not fit Java programming style, e.g., they are not object oriented, unsafe in many respects, and they 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 allows omitting the JNI layer and accessing directly the underlying computing platform. This approach is promising as it leads to much more efficient interfaces than those employing JNI. Unfortunately, it also bounds the resulting code to a particular architecture (currently being only Intel x86) and therefore not very flexible. In contrast, our approach is solely based on JNI and thusly it retains high level of portability.

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

An alternative way of wrapping legacy code is to use *shared stubs* [18]. This technique allows calling an arbitrary function residing in a shared library. It uses system-level routines for the dynamic linking. Such a method is, again, platform-dependent and introduces substantial overhead 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. Fig. 1 shows interactions between a Java application, Java VM, JNI and native code in the case when native methods serve as an interface between the Java application and the legacy native library.

The essential feature of JNI is that it allows a native code to have the same functionality as a pure Java code. In particular, it provides means to create, inspect 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 what 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 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. JNI specification does not define the behavior of a program which fails to release an array. In addition, access functions to arrays over different types have different names. The use of improper functions causes runtime errors rather than compilation warnings. (See Fig. 2a.)

- 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 method 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. 2b.)

- 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 signature depends on the types of all its parameters, what makes a 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. 2c.)

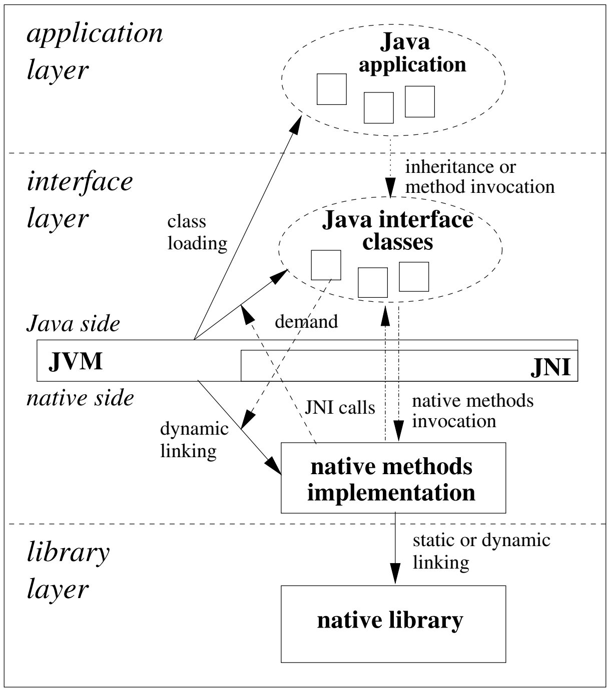

Figure 1: Use of a native library in a Java application

- 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. 2c.)
- 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. 2d.)

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** extension 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: an access to Java fields and variables, invocation of Java methods, use of Java monitors, exception handling, etc. These operations can be carried out with an ordinary Java language syntax – no cumbersome JNI function calls are required.

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

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: determines necessary type signatures, chooses JNI functions to call, loads Java classes, obtains field and method descriptors, performs array and string lock and release operations, handles and propagates Java exceptions, matches monitor operations.

A simple example of canonical "Hello World" program, which 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.

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

(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); }
```

Figure 2: Examples of native methods using JNI features

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

The translation process generates three source files in this case: first of them contains stripped Java source with only a native method declaration, second one contains native method implementation, the third one 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 displayHelloWorld();
    public static void main(String args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

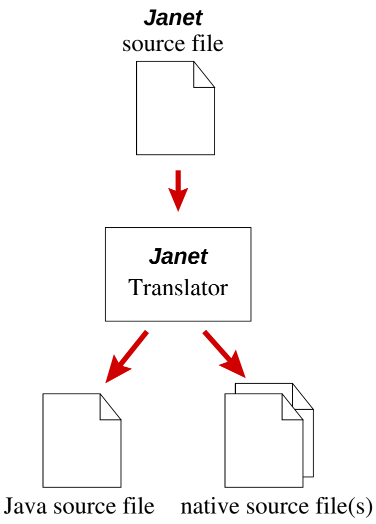

Figure 3: **Janet** translation process

- File `HelloWorldImpl.c`:

```
#include <janet.h>
#include <stdio.h>
Janet_HelloWorld_displayHelloWorld(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 the `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 **Janet** preprocessor using the following naming convention: if the class appears in 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 consider adding 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 a 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 native 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 RuntimeException(); }
    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`) which contains three invocations of Java methods. **Janet** translates these pieces of code into 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 `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 as a parameter the value of some native variable. 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 name space. This is done with a `#(`*expr*`)` syntax in the following example:

```
native void foo() {
    int i=0;
    `bar(i, 0)`;    /* compilation error: what is '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. Large number of native libraries operate on arrays and they could be successfully used in Java when the appropriate interface is provided. For this reason, we considered it to be 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 native 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 simple exposes the array as it is. If the 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`, ...);
}
```

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 losses 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.

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)

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

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:

[3] The address-fetch operator `&` cannot be used with arrays of reference types, what includes all multidimensional arrays.

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

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

Additionally, such variables give explicit control over occurrence 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, which basically means after most of the JNI calls, and provide code to handle them. As exceptions should usually break normal flow of program execution, it becomes easy to mismatch array or monitor lock/release operations in an exception handling code. In contrast, **Janet** provides convenient syntax for exception handling. It is done by adapting Java's exception model and employing `try`, `catch`, `finally` and `throw` statements. The following example shows these concepts:

```
native void method() {
    `try {
        callback();
    } catch (Throwable e) {
        `JNI_EXCEPTION_DESCRIBE();`
        throw new IllegalArgumentException("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 piece of native code here is the `JNI_EXCEPTION_DESCRIBE` macro call – the rest is Java code that handles exceptions. Such a syntax simplification is possible because **Janet** allows to merge subsequent embedded Java operations together which eliminates extra back-tick delimiters. (Compare this example with analogous JNI code shown in Fig. 2c.)

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. It is achieved by use of the 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. Firstly, during compilation, it prevents type checking for the returned value and can thus potentially lead to runtime errors. Secondly, it prohibits **Janet** from executing `finally` clauses as it was required in the example above.

6.5 Unconditional Branch Statements

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

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

The **Janet** preprocessor does not recognize semantics of `break` statement and inserts a monitor unlock code at the end of the block. Therefore, this code unlocks 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 `longjmp()` function call, to be used in native code if they would bail out of the block in which they appear. Also, the use of `return` statement is strongly discouraged for the reasons described in section 6.4.

This issue is related to the C language only. With C++, it is possible to avoid this problem using object destructors.

7 Portability

One of the main goals of the **Janet** project is to retain 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 project 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 starting from version 1.1, so 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.

8 Performance Results

JNI provides a highly portable and abstract interface layer, e.g., it makes no restrictions as to how Java VM represents objects internally. As this approach enables 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 observe it, we performed a series of benchmarks on different platforms.

Table 2 shows performance results for HotSpot Java VM from Java 2 Standard Edition v1.3 for Solaris. The host platform was 4-processor Sun Enterprise 450 with 4 UltraSPARC 400MHz CPUs with 4MB of ECache and 1280MB 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 PC with Pentium II 440MHz CPU and 128MB 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. Number of iterations was chosen empirically (from the range of 10^3 to 10^8) for each test, to assure 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.

At the beginning, the efficiency of both private and virtual native method calls was measured as these are the basis of any native code interface. Next, series of tests were 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**. Next, the performance of Java array access from within a native code was measured for JNI and **Janet**, using both traditional `Get<type>ArrayContents` JNI routines (<u>normal</u>) as well as the `GetPrimitiveArrayCritical` routine (<u>fast</u>) introduced in JNI 1.2. Finally, the additional **Janet**-specific method invocation overhead was measured in the case when arrays of primitive type or `synchronized` statements are used, and when the method must handle Java exceptions, e.g., when there are any callback method invocations.

	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

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.	0.1+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 3: Performance results on Linux OS with JDK 1.3 and HotSpot VM (time shown in microseconds)

	Java	JNI	Janet
private native method inv.	0.55+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

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

As might have been expected, obtaining Java functionality from a 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 the total native method execution time. Therefore, the overall JNI performance seems to be adequate for most applications. However, there are several issues that one has to be aware of (aside from the trivial fact that the additional invocation overhead can exceed any native code performance benefits for native methods with very small amounts of computation and therefore excessive callbacks from native methods should be used with caution):

- 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.
- 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 much slower 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.

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

In most cases, **Janet** adds no more than 20% to the JNI overhead. The additional time is spent to retain the safety of the running code, e.g., for method invocations, **Janet** checks if they did not result in an exception (note that every Java method not declared to throw exceptions may still throw `RuntimeExceptions` and `Errors`). The notable difference between JNI and **Janet** performance is visible only for the array accesses. This is because **Janet** invokes additional method to avoid aliasing problems with multiple references pointing to the same array. This initial overhead, however, would usually be alleviated by the time of actual array processing.

The **Janet** project was originally developed as a Java interface to the `lip` programming library [5, 4, 3]. The `lip` library is built on top of MPI [19, 8, 17] and supports both in- and out-of-core (OOC) parallel irregular problems [23, 2] (i.e. problems that indirectly access large data arrays). To test the performance of **Janet**, a generic irregular OOC problem has been written in Java. Its scalability in comparison to C version is presented in Figs. 4 and 5. All computations in the Java test code were performed at 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.

These results prove 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 described a new approach to creation of Java's interfaces to native codes. The presented Java language extensions and **Janet** preprocessing tool enable simple, fast and convenient development of efficient interfaces while retaining full control over their low-level behavior. At this point, our goal is to provide a visual environment (with graphical user interface) to enable the user to graphically design the structure of Java wrappers for a native library. As a result, the tool would generate **Janet** code which could be further refined by the

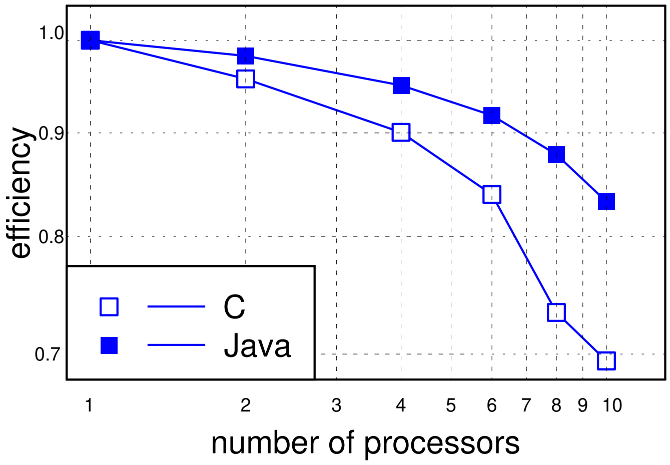

Figure 4: Performance results of the code using the `lip` library from C and Java (`n`= 100)

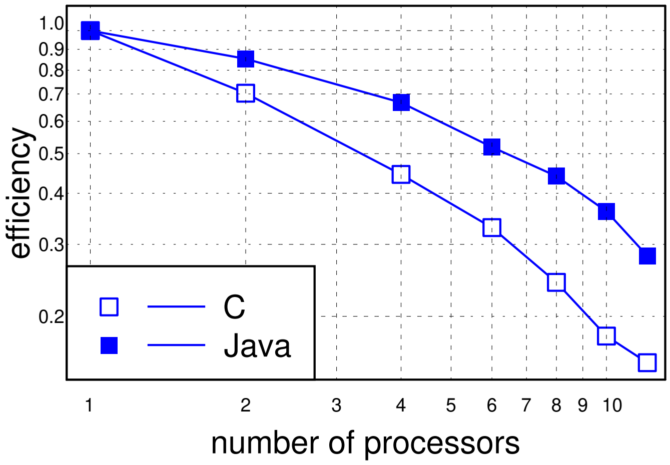

Figure 5: Performance results of the code using the `lip` library from C and Java (`n`= 1000)

user. The fully automatic wrapper generator is also under consideration with its output being subject to potential refinement by the GUI tool. At the same time, we intend to apply **Janet** to enable usage of native resources in the Harness Metacomputing Framework [20, 10]. A support for native languages other than C is also considered. The first candidate here is C++ as it would eliminate the aforementioned problems with unconditional branch statements.

10 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.

[3] M. Bubak, D. Kurzyniec, and P. Łuszczek. Creating Java to native code interfaces with Janet extension. In M. Bubak, J. Mościński, and M. Noga, editors, *Proceedings of the First Worldwide SGI Users' Conference*, pages 283–294, Cracow, Poland, October 11-14 2000. ACC-CYFRONET.

[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.

[5] M. Bubak and P. Łuszczek. Towards portable runtime support for irregular and out-of-core computations. In J. Dongarra, E. Luque, and T. Margalef, editors, *Proceedings of 6th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science*, pages 59–66, Barcelona, Spain, September 26-29 1999. Springer.

[6] O. P. 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*. Addison-Wesley, second edition, 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*, volume 1332, pages 135–142, Berlin-Heidelberg, 1997. Springer-Verlag.

[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, Mar. 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*, 12:519–538, Dec. 1999. Special Issue on Java for High-Performance Applications. Available at `http://www.cs.berkeley.edu/~mdw/papers/jaguar-journal.ps.gz`.

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