Wrapper Generator using Java Native Interface

V.S.Vairale[1] and K.N.Honwadkar[2]

[1]Department of Computer Engineering, AISSMS College of Engineering,
Pune University, Pune, India
vaishali.vairale@gmail.com

[2]Department of Computer Engineering, D.Y.Patil College of Engineering,
Akurdi, Pune University, Pune, India
knhonwadkar@yahoo.co.in

Abstract

The purpose of this paper is to provide a complete automated solution to the wrapping and compilation of legacy code in order to facilitate the use of native libraries in effective ways through Java Native Interface. Legacy codes are those libraries, command line applications or other types of systems that were developed in technologies older than currently used in modern computing environments. Modern software engineering concepts, like software as a service, allow the extension of the legacy code lifetime and the reduction of software maintenance costs. The transformation of a legacy code into a service is not straightforward task, especially when the initial code was designed with a rich user interface. This paper describes a process for the semi-automatic conversion of numerical and scientific routines written in the C programming language into computational services that can be used within a distributed service-oriented architecture such as that being adopted for Grid computing.

Keywords

Legacy Code, AWGNL, C Wrapping, JNI, Data Mediation, Wrapper Generator

1. Introduction

The purpose of this paper is to provide a complete automated solution to the wrapping and compilation of legacy code. Java's object-oriented features, platform independence, and numerous APIs for tasks such as network programming, XML processing, and GUI building, make it a powerful and increasingly popular language for developing Grid-based e-Science applications. However, the task of manually converting the large body of existing high quality, validated code to Grid-enabled, Java-based services is both daunting and expensive. The main aim of the work described is to make this conversion task as automated as possible in order to facilitate the use of such services in composing Grid-based scientific applications.

Recently, Scientific and Engineering communities are employing Grid enabled software applications. To be widely adopted, Java applications in particular will require more support the integration of legacy applications. C and C++ programming languages are used extensively for scientific applications and consequently, the requirement for a method to incorporate software and applications in C or C++ is of prime importance. Two separate tools carry out the wrapping and data mapping functions. The Automatic Wrapper Generator for Native Libraries (AWGNL) tool automatically wraps C routines as Java code using the Java Native Interface (JNI)[6]. This approach can be applied to individual C routines, or to whole libraries, and is based on the routine interfaces given in the C header files. AWGNL allows users with no knowledge of JNI to quickly and easily build a Java wrapper for C routines.

The approach taken is similar to that in the Java-C Interface (JCI). However, AWGNL extends the JCI functionality to deal with arrays of structures, and provides a graphical user interface

through which users can wrap legacy C libraries. Developing new codes, incrementing applications with middleware specific interfaces, or designing applications to explicitly take advantage of distributed resources is a significant burden for the developers who are often reluctant to allocate sufficient effort on non application specific problems. The middleware is therefore expected to ease legacy codes migration to service-oriented infrastructures by proposing a non-intrusive interface to existing legacy codes, and optimizing the execution of the application on the available resources. In this context, enabling legacy code execution on service-oriented infrastructures is a high priority challenge.

The JNI is used to write native methods to handle situations when an application cannot be written entirely in the Java programming language such as when the standard Java class library does not support the platform-specific features or program library. It is also used to modify an existing application, written in another programming language, to be accessible to Java applications. Many of the standard library classes depend on the JNI to provide functionality to the developer and the user, e.g. I/O file reading and sound capabilities. Including performance- and platform-sensitive API implementations in the standard library allows all Java applications to access this functionality in a safe and platform-independent manner. Before resorting to using the JNI, developers should make sure the functionality is not already provided in the standard libraries.

This document describes a process for the semi-automatic as well as automatic conversion of numerical and scientific routines written in the C or C++ programming language into Java programming language. This process involves development of JACAW (Java C Automatic Wrapper), a wrapper tool based on the Java Native Interface (JNI) that can automatically generate the Java interface and related files for any C routine, or library of C routines.

2. Related Approaches

Two main approaches have been adopted in applying Java to Grid computing.

- if the number of the legacy code functions are very high (of thousands order) the available tools for handling services are not facing the requirements.

- the public expose of all the legacy code functions can be a danger for the system hosting the wrapped software if the exposed function list includes functions that modify the host environment.

A first class of techniques comprises the black-box reengineering techniques which integrate systems via adaptors that wrap legacy code as a service (as mentioned above). A second class comprises white-box methods which require code analysis and modification in order to obtain the code components of the system to be presented as services. Both approaches are valid in different circumstances, depending on factors such as the granularity of the code, the assumed users and application area. The first class is mainly applied in the case when the code is not available. Recent papers on this subject are [15] and [8]. A solution for the particular case of interactive legacy systems is described in [13].

Java wrapping can be used to generate the service interfaces automatically as outlined in [1,12]. Prominent examples in this direction are JNL, JAMA, SWIG, JACAW or MEDLI [12]. The most remarkable non-invasive solution is represented by GEMLCA, the Grid Execution Management for Legacy Code [15]. A drawback is that it supposes that the legacy code is activated in a command-line style and does not exploit the possible successive interactions. The same comment is valid also for O'SOAP [13] that also allows legacy command-line oriented applications to be deployed as Web services without any modification, as well as for OPAL [14].

The following two main approaches have been adopted in applying Java to numerical computing. In the first approach, scientific packages previously written in C, C++, or FORTRAN are completely rewritten in Java. Examples include commercial packages such as JNL from Visual Numerics3, and packages from research projects such as JAMA designed by Joe Hicklin, Cleve Moler, Peter Webb (2000).

Work on the Numerically Intensive Java (NINJA) supports the view that there are no fundamental technical reasons why Java should not be used for high performance numerical computing. NINJA uses language and compiler techniques to address Java performance problems, and this type of approach is essential if Java is to be adopted throughout the high performance computing community.

In the second approach, legacy packages are retained and JNI is used to integrate native methods (e.g., C code) with Java. This may not always be an optimal or elegant solution, but it is necessary when large scientific libraries are not immediately available in Java. Wrapping legacy C code can also result in better performance than pure Java code, though this benefit will presumably diminish as a variety of compiler and runtime techniques continue to close the performance gap between C and Java code.

Examples of the wrapping approach include a Java interface to MPI [3] and openSSL [5]. The Janet [2] (JAva Native ExTensions) project makes use of Java language extensions and a preprocessing tool to develop Java interfaces to native code by automatically generating JNI code. Santa Fe (2000) describes a new transformation called alias versioning that takes advantage of the simplicity of pointers in Java. This transformation, combined with other techniques that we have developed, enables the compiler to perform high order loop transformations (for better data locality) and parallelization completely automatically. This compiler is the first to have such capabilities of optimizing numerical Java codes.

This approach requires the source code, but this has the advantage of allowing a high degree of control over the low-level behaviour of the native code. The Jaguar project developed by Johannes Gehrke and Philippe Bonnet (2001) avoids the use of JNI in accessing native code by extending the Java runtime environment to enable direct Java access to operating system and hardware resources. This avoids the need to copy data between the Java and native code, and leads to efficient code, but the approach is architecture specific.

Simplified Wrapper Interface Generator (SWIG) developed by Dave (1995) connects programs written in C (and C++) with a variety of high-level programming languages (including Java). It processes an interface file that defines all of the variables and functions that need to be accessed from Java (or any other language), and generates the JNI interface to the C code for Java. However, SWIG is not completely automatic (the interface file needs to be written) and furthermore, sufficient knowledge of variables and functions may not possible without the source code (which is often not available). In April 7, 2008. SWIG-1.3.35 released.

2.1 Proposed Approach

Any Legacy system has the following characteristics:

• Legacy systems have some reusable and reliable functionality embedded with valuable business logic;

• The functionalities within legacy components are meaningful and more powerful to be exposed in Grid environment from the requirements point of view;

• Reusable components extracted from a legacy system are fairly maintainable compared to maintain the whole legacy system;

• Some components of the target system run on different platforms or vendor products.

The aim of this approach is to reuse recovered legacy components in a framework. Firstly, an evaluation of legacy systems is performed to confirm the applicability of this approach. Secondly, the legacy system is decomposed into component candidates via hierarchical clustering techniques, which are the essential techniques used in component mining in our approach. Then static program slicing techniques are applied to further understand these component candidates. Reverse engineering techniques play an important role in this analysis process. Based on the comprehension, these component candidates are extracted as concerned legacy code segments. In order to be reused as components, the extracted legacy code is refined and encapsulated mainly through JNI (Java Native Interfaces). After these legacy components are created, they will be integrated in a framework as a set of software resource services, which may be composed together to form a grid application.

3. Legacy Code Generator

The command line description has to be complete enough to allow dynamic composition of the command line from the list of parameters at the service invocation time and to access the executable and input data files. High performance legacy codes are pre-existing codes, mostly in C or FORTRAN, that possess the following features: (1) they are domain specific; (2) they are hard to re-use in other applications; (3) they are still useful; and, (4) they are often large.

3.1 The Implementation of the Wrapper Generator

Automatic Wrapper Generator Native Libraries (AWGNL) can be used to wrap existing legacy code in C as a Java code that calls the original code through the Java Native Interface. AWGNL can be applied automatically to wrap entire software libraries thereby saving time and substantially reducing the likelihood of introducing coding errors. AWGNL is based on the Java Native Interface (JNI) [6], which is an API that allows Java code to interact with code written in another language.

AWGNL shields the user from the details of JNI, and does not require the user to have any knowledge of how to use it. AWGNL takes the C header files as input and automatically creates the corresponding Java and C files needed to make native calls. AWGNL also automatically compiles the Java files, creates the header files, and builds a shared library of the JNI-enabled C routines.

Legacy Code Wrapper Generator facilitates the automatic incorporation of a wide range of existing legacy codes. When using it, developers only need to specify the parameters (properties) of the legacy code they want to wrap, then submit the parameters to WG which then generates all the interfaces needed to convert the legacy code into a component.

3.2 Wrapper Generator Data Flow

Wrapper generator facilitates the automatic incorporation of a wide range of existing legacy codes. When using it, developers only need to specify the parameters (properties) of the legacy code they want to wrap, then submit the parameters to WG which then generates all the interfaces needed to convert the legacy code into a component. The interfaces include a IDL interface, an XML definition, an implementation code (Body), a Listener and a Publisher. The Listener and Publisher are used to interact with other components. Figure 1 shows the data flow in the Wrapper generator.

Developers are different from end users in that developers create components, whereas end users make use of the components to construct applications. Developers need to know some information about the legacy code, such as its input(s)/output(s). However, they do not need to know the exact implementation of the legacy code. The main constraints of a legacy code that can be wrapped as a component with the WG are:

- The legacy code can be a sequential code or a parallel code using MPI.
- The legacy code can be written in C, Fortran or Java.
- The legacy code can be located anywhere within a distributed computing network.
- The legacy code must be a binary code and can perform certain functions with some input(s)/output(s).

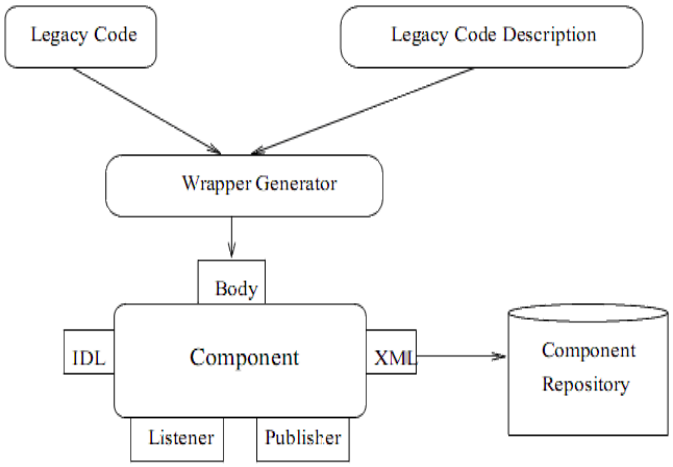


Figure 1. The Wrapper generator data flow

3.3 Data Mediation Interface

The data mediation interface automates the process of mediating the data types between the Java classes and the C function calls. This is accomplished by providing a graphical user interface (GUI) that takes the user through a set of steps to select the parameters for the wrapped function from an input class and then to initialize and set the data for the output class. The use of Data Mediation Interface is divided into three phases:

- DMI displays the instance variables of the classes in the arguments passed to the target routine. The user then graphically maps some or all of these variables to the instance variables of a selected Java class through its get and set methods, thereby creating a port for the target routine. This process of port creation is repeated until all instance variables passed through the target routine arguments have been mapped to a port.

- Next the user mediates the data returned from the target function to one or more Java output classes, thereby creating one or more output port(s). This process is very similar to the mapping of the instance variables of the target routine arguments.
- Finally, the user mediates any other data between the input Java class and the output Java class (es) in phase 2. This is data that is not needed by the target routine, and hence is not passed to it in its arguments, but which is needed in the output port(s) created in previous phase.

4. Role of Java Native Interface

When the Java platform is deployed on top of host environments, it may become desirable or necessary to allow Java applications to work closely with native code written in other languages. Programmers have begun to adopt the Java platform to build applications that were traditionally written in C and C++. The JNI is a powerful feature that allows you to take advantage of the Java platform, but still utilize code written in other languages. As a part of the Java virtual machine implementation, the JNI is a *two-way* interface that allows Java applications to invoke native code and vice versa. Figure 2 illustrates the role of the JNI.

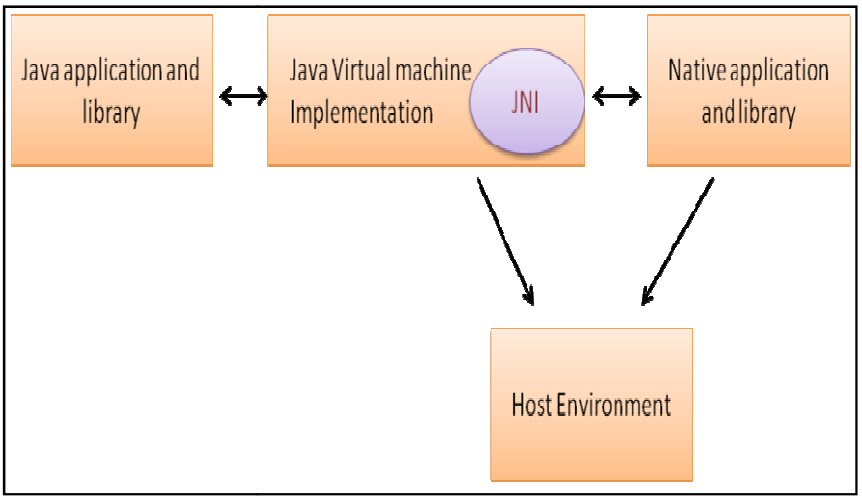


Figure 2. Role of JNI

The JNI is designed to handle situations where you need to combine Java applications with native code. As a two-way interface, the JNI can support two types of native code: native libraries and native applications.

- Use the JNI to write native methods that allow Java applications to call functions implemented in native libraries. Java applications call native methods in the same way that they call methods implemented in the Java programming language. Behind the scenes, however, native methods are implemented in another language and reside in native libraries.
- The JNI supports an invocation interface that allows you to embed a Java virtual machine implementation into native applications. Native applications can link with a native library that implements the Java virtual machine, and then use the invocation

interface to execute software components written in the Java programming language. For example, a web browser written in C can execute downloaded applets in an embedded Java virtual machine implementation.

There is a simple example of using the Java Native Interface. We will write a Java application that calls a C function to print "Hello World!" The process consists of the following steps:

- Create a class (HelloWorld.java) that declares the native method.
- Use javac to compile the HelloWorld source file, resulting in the class file HelloWorld.class. The javac compiler is supplied with JDK or Java 2 SDK releases.
- Use javah -jni to generate a C header file (HelloWorld.h) containing the function prototype for the native method implementation. The javah tool is provided with JDK or Java 2 SDK releases.
- Write the C implementation (HelloWorld.c) of the native method.
- Compile the C implementation into a native library, creating HelloWorld.dll or libHelloWorld.so. Use the C compiler and linker available on the host environment.
- Run the HelloWorld program using the java runtime interpreter. Both the class files (HelloWorld.class) and the native library (HelloWorld.dll or libHelloWorld.so) are loaded at runtime.

Wrapper Generator shields the user from the details of JNI, and does not require the user to have any knowledge of how to use it. JACAW takes the C header files as input and automatically creates the corresponding Java and C files needed to make native calls. JACAW also automatically compiles the Java files, creates the header files, and builds a shared library of the JNI-enabled C routines. Following example shows of what Wrapper Generator does.

4.1 Declare the Native Method

You begin by writing the following program in the Java programming language. The program defines a class named HelloWorld that contains a native method print.

```
class HelloWorld {
private native void print();
public static void main(String[] args) {
new HelloWorld().print();
}
static {
System.loadLibrary ("HelloWorld");
}
}
```

The HelloWorld class definition begins with the declaration of the print native method. This is followed by a main method that instantiates the HelloWorld class and invokes the print native method for this instance. The last part of the class definition is a static initializer that loads the native library containing the implementation of the print native method.

There are two differences between the declaration of a native method such as print and the declaration of regular methods in the Java programming language. A native method declaration must contain the native modifier. The native modifier indicates that this method is implemented in another language. Also, the native method declaration is terminated with a semicolon, the statement terminator symbol, because there is no implementation for native methods in the class

itself. We will implement the print method in a separate C file. Before the native method print can be called, the native library that implements print must be loaded. In this case, we load the native library in the static initializer of the HelloWorld class.

The Java virtual machine automatically runs the static initializer before invoking any methods in the HelloWorld class, thus ensuring that the native library is loaded before the print native method is called. We define a main method to be able to run the HelloWorld class. HelloWorld main calls the native method print in the same manner as it would call a regular method. System.loadLibrary takes a library name, locates a native library that corresponds to that name, and loads the native library into the application. We will discuss the exact loading process later in the book. For now simply remember that in order for System.loadLibrary ("HelloWorld") to succeed, we need to create a native library called HelloWorld.dll on Win32, or libHelloWorld.so on Solaris.

4.2 Compile the HelloWorld Class

After you have defined the HelloWorld class, save the source code in a file called HelloWorld.java. Then compile the source file using the javac compiler that comes with the JDK or Java 2 SDK release:
javac HelloWorld.java

This command will generate a HelloWorld.class file in the current directory.

4.3 Create the Native Method Header File

Next we will use the javah tool to generate a JNI-style header file that is useful when implementing the native method in C. You can run javah on the HelloWorld class as follows:

javah -jni HelloWorld

The name of the header file is the class name with a ".h" appended to the end of it. The command shown above generates a file named HelloWorld.h. We will not list the generated header file in its entirety here. The most important part of the header file is the function prototype for Java_HelloWorld_print, which is the C function that implements the HelloWorld.print method:

JNIEXPORT void JNICALL
Java_HelloWorld_print (JNIEnv *, jobject);

Ignore the JNIEXPORT and JNICALL macros for now. You may have noticed that the C implementation of the native method accepts two arguments even though the corresponding declaration of the native method accepts no arguments. The first argument for every native method implementation is a JNIEnv interface pointer. The second argument is a reference to the HelloWorld object itself (sort of like the "this" pointer in C++). We will discuss how to use the JNIEnv interface pointer and the jobject arguments later in this book, but this simple example ignores both arguments.

4.4 Write the Native Method Implementation

The JNI-style header file generated by javah helps you to write C or C++ implementations for the native method. The function that you write must follow the prototype specified in the generated header file. You can implement the HelloWorld.print method in a C file HelloWorld.c as follows:

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"
JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)

{
printf("Hello World!\n");
return;

}
```

The implementation of this native method is straightforward. It uses the printf function to display the string "Hello World!" and then returns. As mentioned before, both arguments, the JNIEnv pointer and the reference to the object, are ignored.

The C program includes three header files:
• **jni.h** — This header file provides information the native code needs to call JNI functions When writing native methods, you must always include this file in your C or C++ source files.

• **stdio.h** — The code snippet above also includes stdio.h because it uses the printf function.

• **HelloWorld.h** — The header file that you generated using javah. It includes the C/C++ prototype for the Java_HelloWorld_print function.

4. 5 Compile the C Source and Create a Native Library

Remember that when you created the HelloWorld class in the HelloWorld.java file, you included a line of code that loaded a native library into the program:
```
System.loadLibrary("HelloWorld");
```

Now that all the necessary C code is written, you need to compile HelloWorld.c and build this native library. Different operating systems support different ways to build native libraries. Following command builds a dynamic link library (DLL) HelloWorld.dll using the Microsoft Visual C++ compiler:

```
cl -Ic:\java\include -Ic:\java\include\win32
-MD -LD HelloWorld.c -FeHelloWorld.dll
```

4.6 Run the Program

At this point, you have the two components ready to run the program. The class file (HelloWorld.class) calls a native method, and the native library (Hello-World.dll) implements the native method. Because the HelloWorld class contains its own main method,

java HelloWorld

You should see the following output:

Hello World!

It is important to set your native library path correctly for your program to run. The native library path is a list of directories that the Java virtual machine searches when loading native libraries.

5. RESULTS & DISCUSSION

The aim of this work is to develop software that can be used to wrap existing legacy code in C as a Java code that calls the original code through the Java Native Interface. AWGNL can be applied automatically to wrap entire software libraries thereby saving time and substantially reducing the likelihood of introducing coding errors. AWGNL is based on the Java Native Interface (JNI) [6], which is an API that allows Java code to interact with code written in another language.

The wrapper would be able to provide the following functions:-

- It has an internal code editor, which provides features like load, save, copy, paste, clear, etc. created using JTextArea control under java swing.
- Compilation and Building of dll files uses VC++ compiler.
- Registration of DLL using windows service (REGSVR32)
- Provides automatic version which accepts simply the c code for which java code is to be generated. Output is .h file, .dll file, register .dll file, java classes for all the functions, compiled java classes for all the .java files, and a java test code to test that all the functions are available in java indeed.
- Provides external code editor interface. It provides the external editor linkage under the software. User can use any external editor to edit/manage the code. Editors like VC++, JCreator, JEdit, NetBeans, etc. can be linked.
- Process Builder interface. This is an interface required to call all the external compilers, builders, linkers from within java. Also it is needed to capture the outputs of all the compilers.
- Execution time comparison. Time obtained from all the previous three modules are displayed at once with different amounts of inputs. (5000, 10000, 50000 random values).

5.1 Automatic Mode

In this mode the existing java code gets compiled, the header file gets created and DLL will be built. All the procedure in manual module will be done automatically to wrap the code and finally java file get executed. With the help of JNI native files which are written in C will get executed in Java environment by writing all method which we have described earlier in section 4.The following steps user has to do instead of writing whole code in Java while using automatic mode. Figure 3 shows how C routines get wrap into java code.

- In automatic mode user write the C function prototype, save that file as header file such as "mynativeheader.h". for example, C prototype for summation of two integers: *int sum (int, int);*
- Write C function definition in edit C module. Save the file and compile it.
- Next wrap the C function code. We get the java code automatically with wrapped routine of C function. Give only call to the C function in java code, compile it and run it.

The Automatic wrapper for java provides interoperation between Java code running on a Java Virtual Machine and code written in other programming languages (e.g., C, C++ or assembly). The project is useful when existing libraries need to be integrated into Java code, or when portions of the code are implemented in other languages for improved performance. The Java Native Interface is extremely flexible, allowing Java methods to invoke native methods. However, this flexibility comes at the expense of extra effort for the native language programmer, who has to explicitly specify how to connect to various Java objects. The project suggests a template-based framework that relieves the native language programmer from most of this burden. In particular, the proposed technique provides automatic selection of the right functions to access Java objects based on their types.

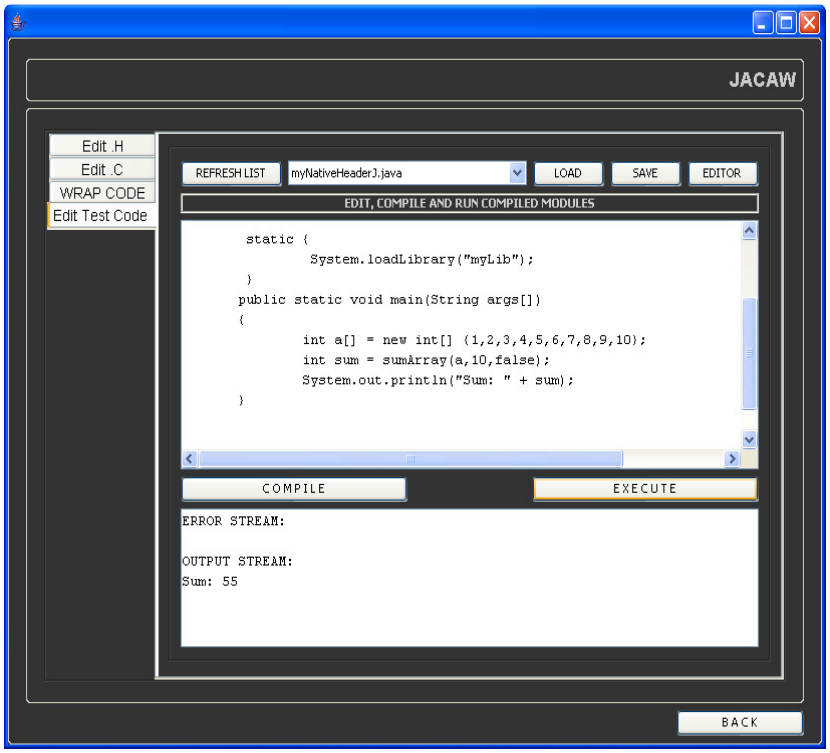


Figure 3. Automatic mode during Execution of Java files with wrap code

5.2 Benchmark Test between C, JNI, Java

Programming through the JNI lets you use native methods to do many different operations. A native method can:

- Utilize Java objects in the same way that a Java method uses these objects.
- Create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks.
- Inspect and use objects created by Java application code.
- Update Java objects that it created or was passed to it, and these updated objects can then be made available to the Java application.

Finally, native methods can also easily call already existing Java methods, capitalizing on the functionality already incorporated in the Java programming framework. In these ways, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them. Figure 4 shows the time comparison for bubble sort by using C, JNI and Java. The following steps show how JNI is more efficient than Java.

- User develops C code for different search algorithms.
- User develops the functions necessary for generating large random input for sorting algorithms.
- Develop code to call these functions in java using JNI
- Calculate execution time taken by every algorithm.
- Display the time taken by every algorithm for the same input.
- Display the time taken by the same code in pure java.
- Display a graph which represents comparison of execution time in JAVA, C and Wrapper Generator system using JNI.

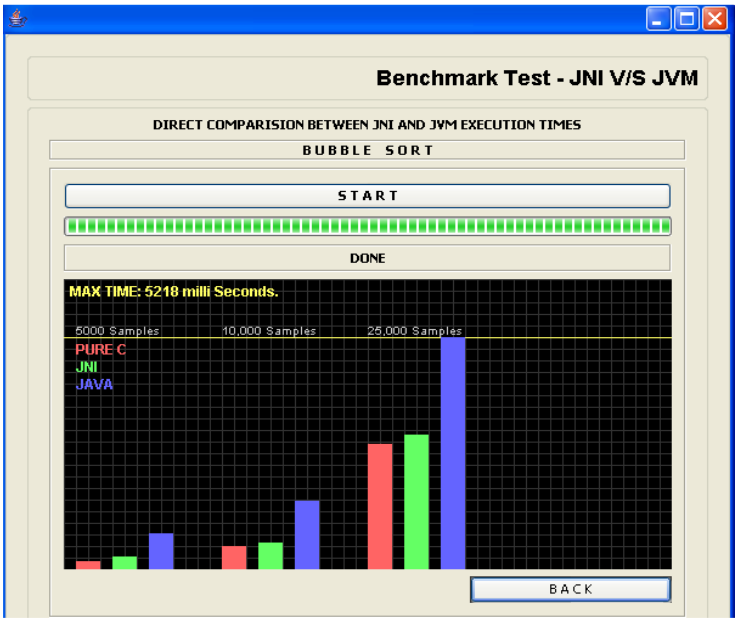


Figure 4. Execution time comparison between C, JNI, and Java code for Bubble sort

6. Conclusion

Wrapper Generator allows programmers to take advantage of the power of the Java platform, without having to abandon their investments in legacy code. It provides a fast and convenient way of enabling legacy C routines to be called from Java applications. It also provides functionality to deal with arrays of structures, and provides a graphical user interface through which users can wrap legacy C libraries. Even conversion of complex data types from Java to JNI and back to Java is done automatically using Wrapper Generator. It provides the best solution to the legacy code problem. And usually it results in better performance than the corresponding Java code. The Wrapper Generator will also be extended in the future to permit wrapped code to be accessed as a Grid service. This will involve generating a description of the service in an XML-based format such as Web Services Description Language (WSDL), and publishing the service in a UDDI and/or Jini registry.

Acknowledgement

I wish to express my deepest gratitude to K.N.Honwadkar for his excellent guidance, invaluable advices and unflinching support without which this manuscript would not have been materialized. I would also take this opportunity to extend my regards to my family for their support and inspiration.

References

[1] D M Beazley, “SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++,” presented at the 4th Tcl/Tk Workshop, Monterey, California, July 6-10, 1996.

[2] J E Moreira, SP Midki®, M Gupta, P Atrigas, P Wu, and G Almasi, The NINJA Project: Making Java Work for High Performance Numerical Computing," Comm. ACM, Vol. 44, No. 10, pages 102,109, October 2001.

[3] K. Pingali, P. Stodgily, A distributed System based on Web Services for Computational Science Simulations, Procs. Of the 20th International Conference on Supercomputing, 2006, pp. 297–306.

[4] M Baker, B Carpenter, G Fox, SH Ko, and X-Y Li, “mpiJava: A Java Interface MPI.” Presented at the First UK Workshop on Java for High Performance Network Computing, September 2002,

http://www.npac.syr.edu/projects/pcrc/papers/mpiJava/mpiJava.pdf

[5] M Welsh and D Culler, “Jaguar: Enabling Efficient Communication and I/O in Java,” Concurrency: Practice and Experience, Vol. 12, No. 7, pages 519–538, May 2000.

[6] M Bubak, D Kurzyniec, P Luszczek, and V Sunderam, “Creating Java to Native Code Interfaces with Janet,” Scienific Programming, Vol. 9, pages 39–50, 2001.

[7] M. Li, , M.S. Shield , O.F. Rana and D.W. Walker. A wrapper generator for wrapping high performance legacy codes as Java/CORBA components in Proceedings of the IEEE/ACM Super Computing'00, Dellas, USA (November 2000).

[8] M. Li, O.F. Rana and D.W. Walker, Wrapping MPI-based legacy codes as Java/CORBA components. Future Generation Computer Systems (FGCS) 182 (2001), pp. 213–223.

[9] R Gordon, "Essential JNI: Java Native Interface", pub.Prentice Hall PTR, 1998. ISBN 0-13-679895-0.

[10] S. Krishnan, B. Stearn, K. Bhatia, K. Baldridge, W. Li and P. Arzberger,, Opal: Simple Web Services Wrappers for Scientific Applications, Procs. ICWS'06, IEEE Computer Press, 2006, pp. 823-832.

[11] T. Glatard, D. Emsellem, J. Montagnat, Generic Web Service Wrapper for Efficient Embedding of Legacy Codes in Service-based Workflows, Procs. GELA 2006, pp. 44—53.

[12] T. Souder and S. Mancoridis. A tool for securely integrating legacy systems into a distributed environment in Proceedings of the Sixth Working Conference on Reverse Engineering (1999) pp. 47–55.

[13] Web Services For Grid-Enabled Problem-Solving Environments, a presentation on the Access Grid on15 March 2002,

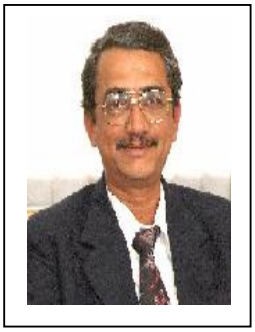
http://www.cs.cf.ac.uk/user/David.W.Walker/WebServicesPSEs.ppt on slide 50

[14] Y. Huang et al., "Wrapping Legacy Codes for Grid-Based Applications", in Proceedings of the 17th International Parallel and Distributed Processing Symposium (Workshop on Java for HPC), 22-26 April 2003, Nice, France. ISBN 0-7695-1926-1

[15] Yan Huang and Qifeng Huang, "GSiB Visual Environment for Web Service Composition and Enactment" Accepted as a poster by the UK e-Science Programme All Hands Meeting 2005, held 19-22 September 2005 in Nottingham, UK.

Authors

K.N.Honwadkar is working with Department of Computer Engineering, D.Y.Patil College of Engineering, Akurdi, Pune, Maharashtra, India. He is Registered Research Scholar from Swami Ramanand Teerth Marathwada University,Nanded,India.
His and research interests include Network Security and Human Computer Interface. He attended many conferences and workshops and published over 25 papers at National / International conferences.



V.S.Vairale is working with All India Shri Shivaji Memorial Society's College of Engineering, Pune, Maharashtra, India. She is pursuing her ME (Computer) degree from D.Y.Patil College of Engineering, Akurdi,Pune. Her Research interests include System Programming, Network Security. She attended many conferences and workshops and published papers.