Automated and Portable Native Code Isolation

Grzegorz Czajkowski, Laurent Daynès, and Mario Wolczko

Automated and Portable Native Code Isolation

Grzegorz Czajkowski, Laurent Daynès, and Mario Wolczko

SMLI TR-2001-96 April 2001

Abstract:

The coexistence of programs written in a safe language with user-supplied unsafe (native) code is convenient (it enables direct access to hardware and operating system resources and can improve application performance), but at the same time it is problematic (it leads to undesirable interference with the language runtime, decreases overall reliability, and lowers debuggability). This work aims at retaining most of the benefits of interfacing a safe language with native code while addressing its problems. It is carried out in the context of the Java™ Native Interface (JNI).

Our approach is to execute the native code in an operating system process different from that of the safe language application. A technique presented in this paper accomplishes this transparently, automatically, and without sacrificing any of the JNI functionality. No changes to the Java virtual machine (JVM™) or its runtime are necessary. The resulting prototype does not depend on a particular implementation of the JVM, and is highly portable across hardware architectures and operating systems. This approach can readily be used to improve reliability of applications consisting of a mix of safe and native code; to enable the execution of user-supplied native code in multitasking systems based on safe languages and in embedded virtual machines; and to facilitate mixed-mode debugging, without the need to re-implement any of the components of the language runtime. The design and implementation of a prototype system, performance implications, and the potential of this architecture are discussed in the paper.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
grzegorz.czajkowski@sun.com
laurent.daynès@sun.com
mario.wolczko@sun.com

Automated and Portable Native Code Isolation

Grzegorz Czajkowski Laurent Daynès Mario Wolczko

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303, USA

1 INTRODUCTION

The growing popularity of the Java programming language [AG98] has not obviated the need for unsafe (native) code. While safe languages offer many benefits, including inherent improved code reliability, increased programmer productivity, and ease of code maintenance, quite often it is desirable to execute user-supplied native methods (referred to simply as *native code*) [Lian99]. There are several good reasons for accepting this impurity: access to devices and programming interfaces to which there is no standard mapping from the language or from the core libraries, direct interaction with operating system services, and possible higher performance of native code. Nevertheless, native code is potentially unsafe and as such may break the contract offered by a safe language.

Ensuring the safety of native code has been the focus of several research projects. The common approach is to let it execute in the same process as the original safe language computation. This can be accomplished by augmenting native code with safety-enforcing software checks [WLA+93], via statically analyzing it and proving it memory safe [NL96], or by designing a low-level, statically typed target language to which native code is compiled [MCG+99]. Although these approaches have their success stories, and at the current state of the art they are practical in many circumstances, their usefulness for addressing problems with an arbitrary native library is rather limited. Ensuring memory safety with these techniques requires source code of native libraries [MCG+99], generating safety proofs [NL96], which is impossible in general, or may incur substantial performance penalties [WLA+93].

The inherent lack of memory safety in native code is not the only problem, however. An equally important issue is guaranteeing the safe use of system resources by independently developed software modules, such as the JVM and arbitrary user-supplied native code. While programmers may have very legitimate reasons to, for instance, customize signal handling in a native method, doing this may interfere with the runtime system of the Java programming language (referred to simply as *the runtime* or *the JVM*).

Directly using certain operating system services by native code may lead to highly unpredictable and difficult to explain behavior, dependent upon the implementation of the runtime and the kind of interference between native code and the runtime. Trapping system calls issued by native code is possible, but deciding whether the call should be allowed to take place or not would depend on the particulars of the runtime. In general, such a control scheme seems likely to be either overly restrictive, preventing some classes of programs from being written, or overly permissive, allowing various collisions and interferences between native code and the runtime of a safe language, or requiring human analysis and manual intervention for each native code module.

Unwanted interference may also occur between two independently written native libraries loaded by the runtime. This means that, for instance, developers of third-party components, coded partly in the Java programming language and partly in C, should considerably restrict their use of system's resources; practicing such "safe driving" is necessary to avoid conflict with other modules, unknown at the time of the development of this particular component, that may be loaded by the same application. The specifics of the "restrict the use of resources" vague guideline may differ among implementations of the JVM which, in general, limits the re-usability of native code. All of the above may conspire to produce unpredictable and difficult to explain behavior, low reliability, and poor software re-usability.

Another concern is raised by embedded JVMs [Morg98]. Any problem mentioned above may not only just crash the application that loaded errant native code, but also abort or distort the execution of the embedding application. Similar issue arises in the context of multitasking JVMs [BG97, HCC+98,Czaj00,BV99,SBB+00,BHL00] where arbitrary independent tasks can execute in the same JVM. The problem is aggravated in this case when native code is written with a single task in mind.

These concerns made us explore the alternative direction – executing user-supplied native code in a different process than that of the JVM. In addition to memory protection via separate address spaces, which guards against memory corruption, this approach prevents conflicts arising from native code abusing resources and interfering with the intended operation of the JVM runtime. The result of this investigation is a design and implementation of a highly portable infrastructure for executing methods from specified native libraries in separate operating system processes.

This work emphasizes that composability of arbitrary unsafe software modules is not just a memory safety issue. The main contribution of the paper is a complete solution, readily deployable in practice. In particular:

- The approach enables safe, reliable, and interference-free composition of native libraries and the JVM runtime.
- JNI is unchanged and programming against it is the same regardless of whether the resulting native code will execute in the same process as the JVM or in another one.
- The resulting infrastructure is independent of the JVM, and does not require changes to implementations of the JVM.
- The infrastructure is highly portable, with a small component that depends on the way arguments are passed to and values are returned from C and C++ methods on a given hardware/software platform, and another part that depends on the IPC mechanism used.
- The actions necessary to execute native libraries in processes other than that of the JVM are fully automated, and require only the libraries themselves, not their source code.
- The performance overhead depends on the ratio of computation carried out by the native methods to the amount of inter-process communication needed and on the efficiency of the IPC mechanism used.

Our initial experience with the prototype system is encouraging, especially considering its transparency to the users and programmers and its seamless interaction with the JVM.

The rest of the paper is organized as follows. Section 2 discusses the problems arising when native code is executed in the same process as the safe language application and the runtime. The advantages and disadvantages of putting native code in a separate address space are discussed in Section 3. Section 4 contains an overview of JNI. The proposed architecture and its implementation are discussed in Section 5. Performance issues are the focus of Section 6. Related work is reviewed in Section 7. Finally, a conclusion section summarizes the paper.

2 MOTIVATION

Most Java runtime environments contain a mix of native code: native code compiled from bytecode, native code that is part of the JVM runtime and interpreter, native code that is part of the core libraries, and, optionally, user-specified native code. This paper discusses only the last of these: user-specified native code, loaded through shared libraries at run time. The other native code is logically part of the JVM runtime, is designed, implemented and tested by the developers of the particular implementation of the JVM, and is totally under their control. None of these properties pertain to user-specified native code. This leads to various problems, described below.

2.1 Conflict of Interfaces

Native code is written against two interfaces: the JNI, which is its sole interaction point with the JVM and the application, and the host operating system (OS) interfaces, involving standard libraries for I/O, threading, math, networking, etc. The latter is also the interface against which the JVM is written, and therein lies a problem. The JVM has to make certain decisions regarding the use of the host operating system interface and of available resources. For example:

- Signal handlers may need to be instantiated to handle exceptions that are part of the operation of the JVM (e.g., to detect null pointer and other memory exceptions, to detect arithmetic exceptions, detect the interrupt signal, etc.).
- The JVM must choose a memory management regime (involving such things as `malloc`/`free` and `mmap`/`munmap`) for its own purposes, including the allocation of thread stacks and red zones.
- Threads accessible in the language are typically mapped onto the underlying system's threading mechanism and a convention is adopted to suspend and resume threads for garbage collection (GC), to assign threads to GC and compilation tasks, to set thread concurrency level, etc.
- The JVM must decide how to manage I/O (e.g., the use of blocking or non-blocking calls).
- The core classes automatically take care of freeing some system resources (e.g., closing open file descriptors); this policy does not extend to the very same resources used exclusively by native code.
- The JVM may have a notion of a current directory and environment variables that should stay unchanged throughout its execution.

Few, if any, of these mechanisms are *composable*, in the sense that it is not possible to take an arbitrary Java program and a user-supplied native library which uses both

JNI and the host OS interface, put them together into an instance of the JVM, put the JVM into an OS process, and expect the resulting system to work correctly. For instance, the native code may set its own signal handlers and clobber those set by the JVM, or it may change the current directory, possibly confusing the JVM. So, in reality, the user-specified native code has to be written to a set of implicit interfaces that do not conflict with the way the JVM uses system resources. These implicit conventions are rarely documented (because they are highly dependent on the implementation decisions within the JVM, which are subject to frequent changes and are usually thought of as private to the JVM), and do not have to be common across even the same vendor's JVMs on the same platform, much less JVMs on differing platforms and certainly not across different vendor's JVMs. Furthermore, it is rare that legacy libraries will respect these conventions; the economics of amending these libraries to respect the conventions are prohibitive (e.g., source code of the libraries may not be available to either the vendor of a particular implementation of the JVM or to the customer using the library). Hence, it may be impossible to use certain libraries in Java applications, or their usability may change with new releases of the JVM. Figure 1 illustrates the conflict of interfaces, and shows how it can be avoided by using another process to host native code.

The problems are exacerbated by so-called "JVM embedding" [Morg98], in which the JVM is treated as a library that can be linked into other applications. In this scenario, it cannot even be mandated that the JVM be in some way "in charge", because it may be subservient to another application. The issue here is that the JVM becomes both the provider of functionality (as an embedded service) and the client of it (when calling native code) and is expected to control native code loaded by itself and at the same time not to interfere with the way the embedding application uses system resources, system interface, etc.

The very same problems are bound to plague emerging multitasking JVMs. Even though the proposed approaches to enable multitasking in the JVM vary [BG97,HCC+98,Czaj00,BV99,SBB+00,BHL00], one theme is common to all these projects: the assumption that no user-supplied native code is run by any of the tasks. This is so because any undesirable operation caused by user-supplied native code (e.g. corrupting memory of other tasks or changing signal handlers previously set up by the runtime) can cause a crash or malfunctioning of the whole JVM, disrupting all the other tasks and defeating all other isolation mechanisms. Unless a comprehensive approach is found to contain various aspects of the damage runaway native code can cause, native code may have to be banned from safe-language multitasking systems.

2.2 Reliability and Resource Guarantees

It goes without saying that the resultant reliability of systems based on this combination is less than desirable. The reliability of the composition of complex applications based on the JVM and native libraries is therefore something of a hit or miss nature.

The JVM needs some resources (file descriptors, memory, etc.) from the underlying OS to perform its function of an ersatz OS for Java applications. However, when arbitrary native code coexists with the JVM, it cannot expect to always find resources available. For example, native code could use up the remaining file descriptors causing failure

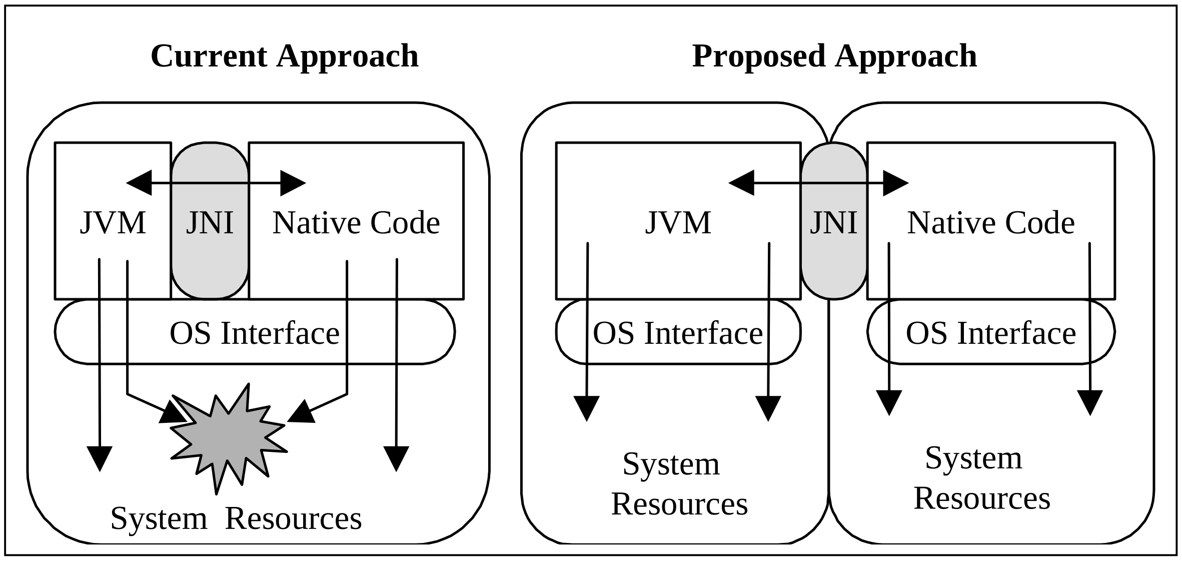

Figure 1. Current JNI implementations do not prevent conflicting use of the OS resources by native code and the JVM (left). The execution of native code in a separate process addresses this problem (right).

in the JVM when it needs to access a file. This happens not only when a Java application opens a file, but also in support of internal operations, such as error logging, class loading, `mmap`-ing memory, etc.

While it is possible to implement the JVM to cope with resource starvation at arbitrary moments, this level of defensiveness requires the JVM to pre-allocate all it needs for its essential operations, artificially inflating the application's usage of system resources. It is also extremely difficult to write and test the JVM code that must deal with all aspects of resource starvation caused by user-supplied native code.

2.3 Debuggability

When there is a problem in the interaction between the native code and the JVM, debugging can be a nightmare. Simple bugs in native code can cause JVM data structures to be corrupted, leading to random failures long after the problem has occurred. If these bugs have pathologies which are time varying, the bugs can manifest themselves in arbitrary places within the JVM or in other native libraries.

For the purposes of fault isolation, it would be desirable if native code bugs were clearly identifiable as such: this would at least save considerable effort. Using techniques enforcing safety at the level of binaries can lead to a clear verdict on whether a particular memory safety violation is caused by a user-supplied native library or by some other part of the runtime. Bugs resulting from conflicting use of system resources are much harder to find. Moreover, the road from detection to finding an actual cause can be a long one, especially when no (or no good) tools for mixed-mode debugging exist for a given hardware/OS/ JVM platform.

The amount of effort needed to track such problems is exemplified by a months-long debugging ordeal by a group of experienced developers, trying to determine why their commercial implementation of the JVM failed when embedded in another application. The reason was that the JVM used signals as its thread suspension technique internally. The action of receiving these signals interrupted pending system calls from the embedding application's non-Java threads. Eventually, the problem was "solved" by not using the signals [Skin00].

3 PROCESS-BASED ISOLATION

The problems identified above indicate that the co-location of user-supplied native code with the runtime of a safe language leads to numerous difficult issues. This section explores the benefits and disadvantages of moving native code out of the JVM's process into a separate process. Because processes of modern operating systems identify memory protection (via hardware-enforced address space separation) with resource allocation and control, they are well suited to address the problems discussed above.

3.1 Advantages

Composability. Separating the JVM from user-specified native code means that the only interface between the two becomes JNI: a standard, well-defined and widely accepted interface. There is then no implicit contract between JVM and native code concerning memory management, threading, signal handling and other issues. This solves the composability problem neatly, both of the JVM and native code and of multiple native libraries. By placing the native code in a separate process it has full control of its own resources: memory, signals, threads, file descriptors, sockets, etc. There are no unexpected interactions with the JVM in these areas.

Fault isolation. A wild pointer bug in native code is not going to corrupt JVM data structures. It is much more likely to lead to a fault within the native code's process, which is then easily attributed to the native code. False JVM bug reports should thus be reduced. Similarly, this approach allows for a quick identification of which native library causes a fault, when more than one such library is used by a Java application.

Resource management. There is no danger that resource management policies will conflict: for instance, the native library can `mmap` at any address it chooses. Further, resource management errors (e.g., careless use of file descriptors, memory, etc.) do not percolate from the library to the JVM.

JVM embedding and multitasking in the JVM. Embedding the JVM within an application should be simple and reliable. With native code moved safely to another process, a well-tested implementation of the JVM is unlikely to crash its host application and yet can offer the benefits of interacting with native code. Multitasking in the JVM benefits in a similar way – one task's native code cannot crash or disrupt any other task, provided each task's native code executes in a separate process.

Interoperability across different address widths. Ensuring that 64-bit JVMs are able to use existing 32-bit native libraries may smooth the transition to the former for many users. The proposed technology can be used to this end, as long as both the JVM and the native code obey JNI to communicate with each other and as long as the chosen IPC mechanism can flawlessly exchange data between the two. Such transparent interoperability across different address widths can also be used to let 32-bit JVMs use 64-bit libraries. This generalizes to arbitrary large future address widths.

Remote processing of native code. Moving native code into a separate address space turns JNI (in addition to its original function) into a specialized, high-level inter-process communication protocol, layered on top of an IPC mechanism already present in the host operating system. This IPC mechanism can be replaced by a networking protocol, enabling the remote execution of native libraries. This can be useful for load balancing or when native libraries and the JVM runtime are written for different hardware/OS platforms. Thus, a technology enabling transparent execution of a native library in a separate process at the same time enables the remote use of the library in a heterogeneous environment, without having to re-code or wrap the library with RMI, CORBA, and such – JNI suffices as the interface between the JVM and native code, regardless of where the execution of either takes place.

3.2 Disadvantages

Performance. The most important disadvantage of this approach is the performance overhead incurred when crossing process boundaries. However, one important observation is that significant IPC overhead may be tolerable. We are only moving user-specified native code into another process; the performance-critical native code in the JVM and core classes would still be in the same process, and its performance would not be affected.

Two facts mitigate this problem. First, a common idiom of caching results of some JNI calls [Lian99] decreases the number of their invocations. Second, research ideas on low-cost IPC have found their way into commercial operating systems. Two examples are *doors* [HK93] available in the Solaris™ Operating Environment and local RPC available on win32-based systems [Roge97]. They were designed to address precisely the problem of high-speed RPC-like communication within a single machine and can be used to send data between the JVM and native code. The overhead of copying a few words of data is typically negligible when compared to the cost of establishing transfer of control. This may not be so for the copying overhead for large data structures, such as arrays. One possible workaround is to use shared memory facilities, which in turn may re-introduce some of the problems of the co-location of native code with the JVM.

For many applications the possible overheads of our approach are a small price to pay for increased reliability. Moreover, with the improving quality of Java compilation techniques, the need to use native code as a performance boosting mechanism will become marginal.

Differing semantics. As the prototype described in Section 5 demonstrates, it is possible to execute native code in a separate process transparently, without the loss of functionality and without any dependence on the JVM. However, it is also possible to write legitimate native code that behaves differently when executed under “traditional”, in-process JNI and under out-of-process JNI. A contrived example showing differing semantics is a program that opens a file using standard `java.io` classes and uses reflection to get the value of the file descriptor of the open file, typically stored in a non-public field of a `java.io.FileDescriptor` object (this requires knowing the name of the field). The value of the file descriptor is then passed to native code, where it is used to access the file system. In this example, the execution of native code in a separate process will almost certainly lead to different results than when all the program code is co-located in the same process. Thus, even though concocting such examples is difficult and requires intimate knowledge of the JVM and its core classes, differing semantics may arise in complex programs. A preventative rule of thumb is to make sure that the lifecycle of each system resource (initialization, use, termination) is confined to either the JVM or user-supplied native code, and never spans the two.

4 OVERVIEW OF JNI

Before moving on to describing our design and implementation, a brief overview of JNI is due (for a complete reference, the reader is referred to [Lian99]). JNI is the sole point of interaction between the JVM and user-supplied native libraries. The interaction can have two forms: *downcalls* (when a Java application calls a native method) and *upcalls* (when a native method needs to access data or invoke methods of the Java application).

Downcalls result in calls to C functions (a C++ mapping also exists), whose names are generated from the names of Java methods declared as native. Upcalls are invoked via a *JNI environment* interface. For example, this class

```
class Test {
   static native int doubleIt(Integer i);
}
```

defines a native method which takes an object of the type `java.lang.Integer` and returns an integer value. The JVM expects a corresponding C function with the following declaration:

```
jint Java_Test_doubleIt(JNIEnv *env,
               jclass cls, jobject iref);
```

When a Java application invokes `Test.doubleIt(i)`, the C function `Java_Test_doubleIt(env, cls, iref)` will be called. The first argument is a pointer to the JNI environment interface, which groups JNI upcall functions. The second argument is a reference to an object representing the class to which the method belongs (`Test` in this case). The third argument is a reference to the argument of the original call. The values of `cls` and `iref` and the data structures they point to depend on a particular implementation of the JVM and should be handled by JNI functions only. These functions are accessible via `env`. To see how it is done, let us take a look at the body of the native method:

```
jint Java_Test_doubleIt(JNIEnv *env,
                jclass cls, jobject iref) {
  jclass intcls =
    (*env)->GetObjectClass(env, iref);
  jmethodID mid =
    (*env)->GetMethodID(env, intcls,
                        "intValue", "()I");
  jint val =
    (*env)->CallIntMethod(env, iref, mid);
  return val * 2;
}
```

By invoking methods on `env`, the reference to the class of `iref` is obtained first, then an identifier of the instance method "`intValue()`" of that object is found, and finally the method is invoked so that its return value can be doubled and returned back to the calling Java method. An important point about the JNI environment interface is that even though meta-data such as references to classes and method identifiers are visible to native methods, they are both *transient* and *opaque*. They are transient because they are obtained only to be passed on to other JNI functions (although they can be cached in native code to avoid repeated lookups). They are opaque because they are meaningless to native code. This simplifies the implementation of our prototype (Section 5), since no native code should use these values for anything else than as arguments to JNI functions.

The JNI environment interface defines more than 220 methods. This large number is a consequence of two facts. First, many operations (invoking a method, accessing a field) have several variants, depending on the type of field or return value of the method, and on whether the field or method is static or not. Second, JNI offers a rich set of interactions with the JVM. In addition to accessing class and object fields and invoking methods, the following are examples of the functionality offered by JNI: monitors available in the Java programming language can be entered and exited, which allows for synchronization between native and Java code; new references to objects can be created so that the GC is aware of them; array regions can be accessed; new objects can be created; reflective capabilities of the Java programming language can be used; exceptions can be thrown and caught by native code.

There are two mechanisms to resolve names and link native methods as corresponding Java methods. The first one requires that implementations of native methods be named according to a specific convention (e.g., `Java_Test_doubleIt`). When the convention is followed, linking a native method is then just a matter of a name lookup in the symbol table of a native library; such lookup is performed automatically by the JVM. The second mechanism does not rely on the JVM to search for the native method in the already loaded native libraries. Instead, JNI programmers can explicitly link native methods with their implementations by registering a C/C++ function address as the implementation of a particular native method. This is convenient especially when a native application embeds a JVM implementation and needs to link with a native method implementation defined in the native application. The names of such native methods of the embedding application almost certainly do not obey the naming convention required by the JVM for implicit linking. This has motivated the second, explicit way of name resolution and linking.

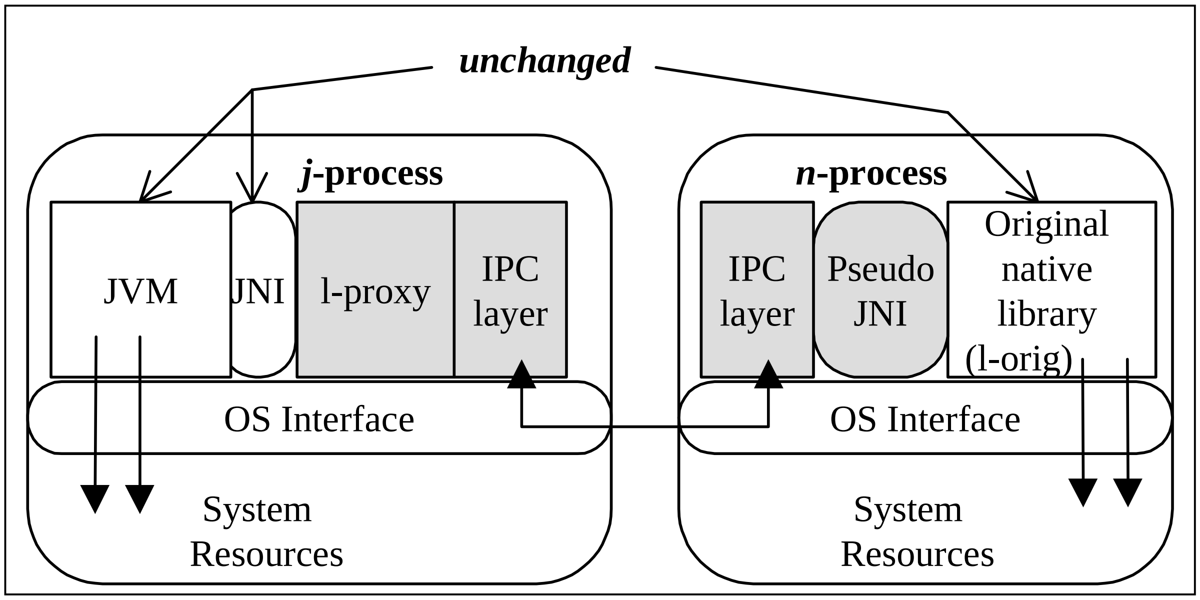

Figure 2. A schematic view of the prototype.

5 THE PROTOTYPE

A highly portable, JVM-independent infrastructure for automated and transparent execution of user-supplied native code in a separate process has been designed and implemented. This section discusses the design and expounds on some more technical aspects.

A high level view of the system is shown in Figure 2. It consists of two processes. The first one, *j-process*, executes the JVM and the Java application. The second one, *n-process*, acts as a server processing requests to execute native methods, and is linked with the original user-supplied native library; let us call it *l-orig*.

In order to achieve JVM-independence, a native library with the same name and exported symbols as *l-orig* must be produced so that it can be loaded by the Java application executing in *j-process*. Let us call this library *l-proxies*. It is generated through an automated analysis of the symbol table of *l-orig*. All names of defined functions starting with `Java_` and `JNI_` are used to generate a source file, in which all these functions are redefined to ship all of their arguments along with an integer uniquely identifying the function to *n-process*. Upon receipt of such a message, *n-process* executes the requested function with the supplied arguments.

Each JNI function takes a JNI environment interface pointer as its first argument. This simplifies the handling of upcalls. Upon receipt of a request, *n-process* replaces the first word in the list of received arguments with its own version of the JNI environment pointer. This *pseudo* JNI environment redefines all JNI functions so that each of them ships all of its arguments along with its unique identifier to *j-process*, where the upcall is dispatched to the original JNI method.

To see the automation and transparency aspects of the system, let us analyze the steps necessary to run the library containing the method `Java_Test_doubleIt` (see Section 4) in a separate process. Let us assume that the native implementation is compiled into a library called `libtest.so`. A set of makefiles and scripts is used to produce `libtest-proxies.so`. This new library contains a function `Java_Test_doubleIt`, which gathers all of its arguments and sends them to *n-process*. The original library (`libtest`) is then linked with a request-processing loop to form *n-process*, while `libtest-proxies.so` is renamed to `libtest.so` so that when *j-process* requests the native library the proxies are loaded instead. An initialization code in `libtest-proxies.so` is responsible for starting *n-process* before any downcalls are issued.

Several details have to be explained before this high-level picture can materialize into a working system. They include handling upcalls, accessing arrays and strings, explicit registration of native methods, issues related to the underlying architecture, and the choice of an IPC mechanism.

5.1 Upcalls that Invoke Methods

Implementing the custom JNI environment interface was straightforward for the JNI functions, which take a fixed number of arguments of primitive types or of types describing or referencing various components of a running Java application. JNI functions not in this category are array and string handling functions (Section 5.2) and JNI functions for invoking Java methods or object constructors, described below.

Invoking a Java method or object constructor via JNI requires passing the arguments for the method. JNI provides three ways to do this: (i) passing arguments using functions with a variable number of arguments (by using the "…" construct of C), (ii) passing arguments as a variable-length list, as a value of the type `va_list`, and (iii) passing a vector of unions of the JNI-defined type `jvalue`, each of which holds one argument for the upcall. Only in the first case it is possible to compute the number of upcall's arguments by analyzing the current stack call frame. However, this is not portable and does not address the remaining two cases.

A portable solution is offered by the design of JNI itself. Before invoking a Java method via a JNI upcall, the method identifier has to be obtained first. This can only be done by calling a JNI function (`GetMethodID` or `GetStaticMethodID`) and supplying it with, among other information, the signature of the Java method or constructor to be executed by the upcall. The signature is an easy-to-analyze string. The analysis is performed right after successfully obtaining the method identifier, and consists of computing the number and size of the arguments. The original method identifier and the argument information are stored in a data structure whose address is returned as the method identifier to the caller (a native method in *n-process*). This data structure is then used to extract the original method identifier and associated method information before forwarding the upcall request to *j-process*.

5.2 Upcalls that Access Arrays and Strings

Another design decision had to do with accessing arrays of primitive types. This can be accomplished in two ways in JNI. The first way is to obtain a copy of the elements (*Get<Type>ArrayRegion* family of upcalls) and then set it accordingly (*Set<Type>ArrayRegion*). The second way is more direct - a pointer to the array elements is obtained (*Get<Type>ArrayElements*) and then released after accessing the elements of the array is complete (*Release<Type>ArrayElements*). Since the direct pointer obtained in *j-process* is meaningless in *n-process*, our implementation of the *GetArrayElements* functions returns a copy of array elements to *n-process*. It also stores the original direct pointer in a hidden prefix of the array. Each time a *ReleaseArrayElements* function is called in *n-process*, the original direct pointer is retrieved from the array prefix and then sent to *j-process* along with the request to release array elements and with the elements themselves. Let us note here that the JNI specification stresses that changes made to the array elements obtained via any of the *GetArrayElements* upcalls will not necessarily be reflected in the original array until a corresponding *ReleaseArrayElements* is called. Thus, our implementation does not violate the semantics as defined by the specification. JNI functions for the direct manipulation of string's characters are handled in a very similar fashion.

5.3 Architecture-specific Issues

The prototype system is highly portable. There are only two parts of it, which in general require code specific to a particular platform: (i) figuring out how many arguments are passed to a downcall in *j-process*, and (ii) returning the correct value after executing the original native method in *n-process*. Both stem from the fact that, in general, neither the number of arguments nor the type of return value of a native method can be inferred from the native library.

In our prototype system, running on the Solaris Operating Environment on a SPARC™ processor, reading the number of words taken up by arguments from a stack frame solves the first issue. This reading is done by the generated code in *l-proxies.* Properly obtaining the return value of a native method is done by a request-processing loop in *n-process* as follows. The return value of a native method can be either of a primitive type or a reference to JVM entity (object, class, method identifier, etc.). In any case, this value is returned in one of two registers: fixed-point `%o0` or floating-point `%f0` [WG94]. After the call, the values of both of the registers are read by *n-process* and then sent back to *j-process*, where they are restored just before returning from the downcall. Thus, without knowing what has actually been returned by the method, this value is properly returned to the caller. There is no danger in overwriting a value of, let us say, `%o0` when in reality a call returns a value in `%f0`, since neither register is preserved across function calls. Other architectures may have different idiosyncrasies with respect to these issues. Some of these problems may not arise with C++ native libraries on some platforms, since it may be possible to extract method signatures from mangled names in library symbol tables.

5.4 Explicit Native Method Registration

This section describes a portable implementation of the explicit registration of native methods (Section 4). A fixed number (256 in the current design) of *anonymous proxies* are linked with *j-process*. An upcall requesting the binding of a Java method (specified as a triple: {name, signature, defining class}) to a particular function address *rFptr* in *n-process* obtains an anonymous proxy *ap*; *ap* is then passed on to the actual JNI upcall in *j-process*. A static variable in *ap* is used to initialize it with the following information: the number of arguments this method should expect during invocations (computed from the signature) and *rFptr*. This

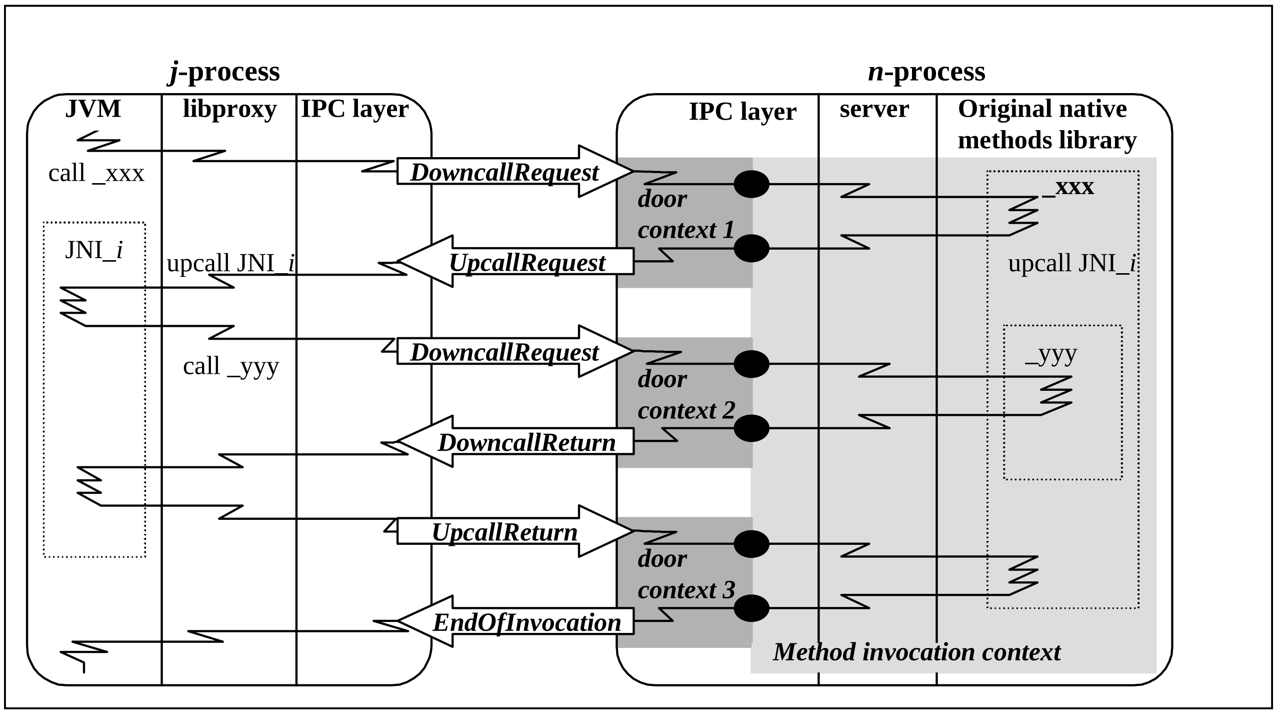

Figure 3. Preserving the context of the JVM thread and the context of native method execution. Black ovals represent *ucontext* swaps.

information is then used during the registered downcall invocations to fetch all its arguments and pass them along with *rFptr* to *n-process*.

This solution is portable but limits the number of explicitly registered native functions. Solutions without this constraint include generating machine-code stubs dynamically (non-portable) or generating and loading new libraries, filled with anonymous proxies (requires operating system-specific code to load a library). An unconstrained and fully portable solution would require JNI to offer a standard way to obtain the name and signature of the currently executing Java method which, from the perspective of native code, is the topmost Java method on the execution stack (currently this information is available via the JVM Debugging Interface [Sun00a], but not via standard JNI). With such a mechanism, only one anonymous proxy would be needed. It would be linked with each explicitly registered native code and, on invocation, it would use the information from the topmost frame on the stack of Java method invocations to compute the size of arguments and to determine the native function to be called via a lookup based on the name of the downcall. Such a mechanism would also address the issues highlighted in Section 5.3.

5.5 Thread-related Issues

An important design question was whether the upcalls should be handled in the context of the thread that originally issued the downcall. Allowing different contexts could potentially simplify the implementation but would lead to various problems. For instance, an exception thrown in an upcall has to be dispatched to the thread that caused the downcall. Similarly, requests to obtain a stack trace should look identical to those generated by traditional JNI systems. This may be hard to achieve if different threads generate a downcall and then handle subsequent upcalls. Another example of problematic behavior is executing `Thread.currentThread().setPriority()` via a sequence of upcalls. Yet another potential problem is caused by monitors acquired by application's threads; handling upcalls in a thread context different from that of the downcall may lead to having to re-acquire monitors and to deadlocks.

In order to avoid these problems, all upcalls are handled in *j-process* by the thread that has issued the corresponding downcall. Hence, a thread in the *l-proxy* code may return from a downcall for two reasons: because the native method completed, or because of an upcall issued by that native method. In the latter case, the *l-proxy* calls the JNI locally, and returns its results to the *n-process*. How upcall requests are delivered to the thread that has invoked the initial downcall is taken care of by the *IPC layer*, described below.

5.6 IPC-specific Issues

The design is independent of any particular IPC mechanism chosen to exchange data between *j-process* and *n-process*, although the chosen mechanism can substantially change the performance of crossing the JNI interface. Our IPC layer isolates the rest of the system from details such as how upcalls are guaranteed to execute in the correct context (Section 5.5). It enables simple replacements of one IPC mechanism with another, simplifying cross-OS porting. Typically, an IPC layer based on mechanisms optimized for the underlying hardware and operating system platform would be favored.

Our implementation of the IPC layer uses doors [HK93], which is an efficient IPC mechanism available on the Solaris Operating Environment and on Linux [Lang98]. Doors achieve low latency by transferring control directly (handoff scheduling) back and forth between the caller's and the callee's threads [MM00]. We have also experimented with a sockets-based implementation, but the overheads were much larger than those of doors (Section 6).

Maintaining the performance advantages of doors handoff scheduling is not straightforward because of upcalls: an upcall from the native code requires a transparent transfer of control back to the original thread without losing the native method execution context on the *n-process* side. Another door call directed at the original JVM thread (from *n-process* to *j-process*) cannot be made, since this thread is already waiting on a door call for the native method. One way to circumvent this problem is to use an intermediate thread to perform the door call for the method invocation so that the original thread can service subsequent door calls from the *n-process*. However, this adds two additional threads for each thread executing native methods; it also adds additional full thread context switches. The alternative, adopted in our design, is to implement upcalls as a return from the original door call.

In order to transparently retain the native method execution context, the standard *ucontext* facility available on modern commercial UNIX® systems is used as shown in Figure 3. A pool of *ucontext* structures is pre-initialized at startup time by *n-process*. Door calls are issued by *l-proxy* for both invoking native methods and for returning results of JNI upcalls. That way, *j-process* is always issuing door calls and the *n-process* is always returning from door calls. The returned information indicates whether it is a return from a downcall or an upcall request. Upcall request messages carry a *ucontext* identifier (currently, a pointer to the actual *ucontext* structure) that is always included in data returned back by *l-proxy* so that the proper context can be re-established in *n-process.* When the last active native method completes, the corresponding *ucontext* is returned to the pool. This solution takes full advantage of the handoff scheduling of doors, and guarantees that a chain of native methods executes with the same stacks and contexts in both the *j-process* and the *n-process*.

5.7 JVM-specific Issues

There are no issues specific to or dependent upon a particular JVM implementation. This is a very important point, worth stressing in boldface: **There are no JVM-specific issues**. In particular, this means that the design presented in this paper can be implemented and used treating the JVM as a black box, and regardless of the version or the vendor of the JVM, as long as it complies with the JNI specification and as long as the chosen communication and threading mechanism does not conflict with the internals of the JVM. We have tested our prototype with three different implementations of the JVM available from Sun Microsystems and encountered no such conflicts.

It is important to note that, depending on the implementation of the JVM, native code contained in *n-process* may still cause a crash or malfunctioning of the JVM executing in *j-process*. This may happen when the JVM does not check whether class and object references and method and field identifiers passed to it via JNI calls are valid. For instance, native code could pass an arbitrary number in place of an expected valid method identifier. JVM implementations with proper checks in place would detect such situations and, for instance, throw an exception or return an error-signaling value. JVMs without such safeguards are vulnerable to these kinds of problems.

5.8 Safe Composition of Multiple Libraries

The description so far assumes that all calls to all native libraries were routed to a single *n-process* server. As mentioned earlier, isolating native methods from different native libraries may also be needed. A simple solution is to create as many *n-process*es as there are user-defined native method libraries. Since the corresponding proxy library creates *n-process*es, routing a native method call to the right *n-process* is straightforward.

To avoid a proliferation of *n-process*es, it may be useful to perform the processing of native methods from multiple libraries by a single *n-process* when isolation between these libraries is not a requirement. Library groups may be described in a simple configuration file, whose location is known to each proxy library.

Another library management issue, not investigated in this paper, is loading the same native library by multiple tasks in a multitasking JVM. Multiple *n-process*es are then acceptable, but more logic has to be put into proxies to multiplex/demultiplex calls based on the task id of the down-caller.

6 PERFORMANCE ISSUES

The purpose of this work is primarily to increase debuggability and reliability of applications consisting of safe and unsafe code. Nevertheless, it is interesting to know the overheads introduced in our prototype. The cost is directly proportional to the number of JNI downcalls and upcalls executed by an application, as these are the only places where any overheads were introduced. They are also dominated by the cost of the IPC mechanism chosen and by the cost of thread context management.

Table 1 summarizes the overheads introduced by our approach. The measurements were obtained on a Sun Enterprise™ 3500 with four UltraSPARC™ II processors clocked at 400MHz running the Solaris Operating Environment, version 2.7. The Java HotSpot™ virtual machine, client version 1.3.1, was used. The “downcall” column reports the cost of a very simple downcall, which takes no arguments and returns immediately, while “downcall + upcall” additionally issues the *GetVersion* upcall, which returns immediately with an integer specifying the JNI version used. The 54µs overhead of a downcall breaks down as follows: 18.5µs is taken by a door call and return, 30.8µs is the cost of two *ucontext* swaps and the rest (4.7µs) are various bookkeeping and data copying overheads. Similar analysis applies to the overheads associated with upcalls. This number would be much higher if sockets were used – a one-word round-trip message takes about 128µs.

The overheads (about 400 times higher than the plain in-proc JNI downcall) seem very large. And they are indeed, for downcalls that do not compute much. To see how quickly the overheads become tolerable, let us analyze a simple program which performs $A = A^2$ for matrix A. The matrix is coded as an array of floating point numbers and squaring it requires one downcall (to initiate the native call) and two upcalls (to fetch the matrix entries and to set the computed result). Figure 4 summarizes the overheads as a function of array size. For small arrays the overheads are prohibitive. For 40x40 arrays they become tolerable, and virtually disappear when the array size approaches 90.

Another insight into the costs is obtained as follows. All upcalls and downcalls performed by SPEC JVM98 [Spec98] benchmarks were counted. These calls were

	JNI in-proc	JNI out-of-proc
Downcall	0.136µs	54µs
Downcall + upcall	0.163µs	106µs

Table 1. Overheads of executing native code in a separate process.

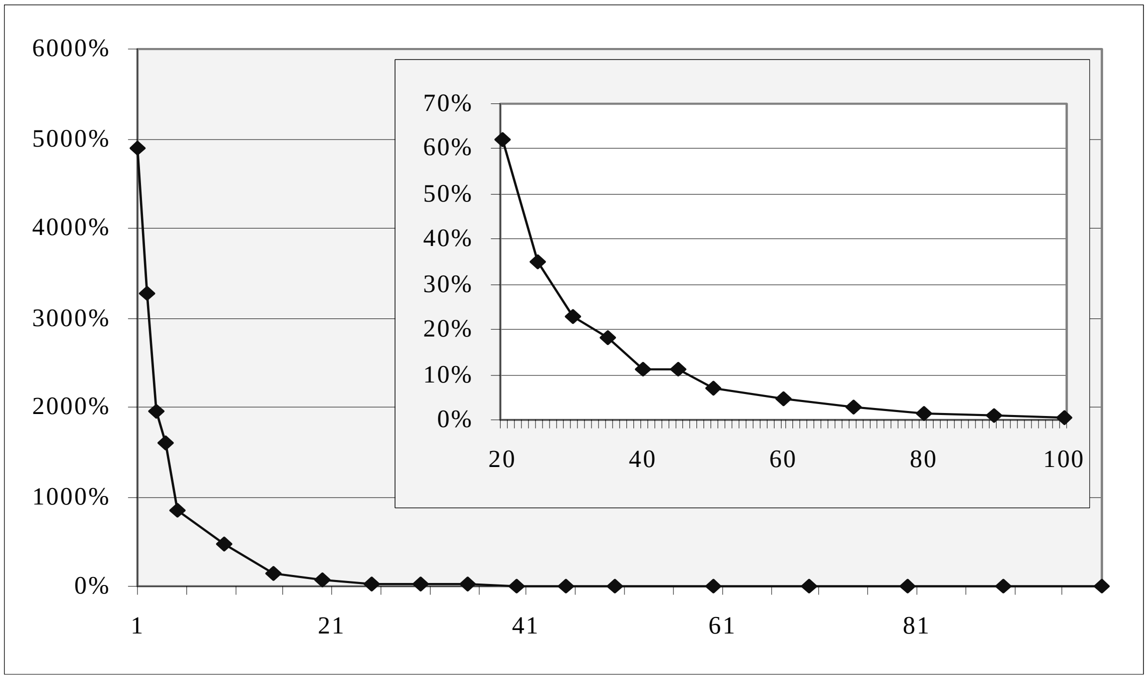

Figure 4. The overheads of computing the second power of a matrix in a native method in a separate process. The overheads are reported relative to executing the method under standard JNI. The overheads are plotted as a function of the matrix size. The matrix size is the number of rows (which is equal to the number of columns) and is shown on the horizontal axis.

caused by core Java classes, since there are no user-level native libraries associated with the benchmarks. The counts were used to estimate the overheads of the scenario in which all native code of core libraries was executed in a separate *n-process*, each downcall adding 56µs and each upcall adding 52µs of performance overhead. The results, summarized in Table 2, show that the number and frequency of native calls issued by core classes varies greatly across the benchmarks, and so would the performance overhead of the proposed isolation. These numbers are no substitute for more realistic benchmarks, and serve only as an evidence of the high dependence of the frequency and number of native calls (and the associated overheads when the JNI isolation scheme is used) on the behavior of a particular application.

Benchmark	Downcalls	Upcalls	Overhead [%]
compress	2267	4911	1.03
db	101963	45127	15.46
jack	2573766	42123	887.88
javac	4204338	46541	680.87
jess	2681855	40509	859.22
mpeg	1252532	14595	170.14
mtrt	192095	7092	40.23
raytrace	169160	7064	37.26

Table 2. JNI downcalls and upcalls caused by core Java classes and the overheads if they were to execute in a separate process.

The main utility of the proposed system is increased debuggability and reliability of native code, and performance issues are typically of secondary importance in such settings. Whenever performance matters and the proposed isolation scheme is preferable over standard JNI, the number of native calls to libraries executed as *n-process*es should be minimized. Also, if the system is used for debugging, a tested library can be moved back into the same process as the JVM, which removes the overheads.

7 RELATED WORK

The efforts to contain native code can be characterized as whether it can run in the same address space as the "main" program or not. Specific solutions in each of these areas are discussed below.

7.1 Controlling Native Code

In order to be able to safely co-locate native code with other application and runtime components, the behavior of binaries must be constrained to ensure memory safety. A dynamic approach is software fault isolation (SFI) [WLA+93], which parses binaries and inserts runtime checks on memory operations. Several flavors of SFI exist, depending on the desired level of memory safety. In the general case, when all read and write operations have to be checked, the overheads of runtime checks can amount to 20% [WLA+93]. SFI toolkits must be very carefully crafted. Otherwise, the overheads may be much higher. For instance, the VINO extensible operating system [SES+96] uses SFI for protecting the kernel from mischievous extensions (*grafts*). The worst-case runtime overheads of applying straightforward (not optimized) SFI to grafts can be as high as 100%.

In contrast to the dynamic approach of SFI, Proof-Carrying Code (PCC) [NL96] advocates static analysis of unsafe code. The code producer is required to generate a formal proof that the code complies with the certain policy. The client (i.e. receiver of the untrusted code) can easily validate the code against the proof and execute it only if the validation is positive. Thus, the runtime costs are negligible. However, the general problem of proving the memory safety of arbitrary binary code is undecidable. Inserting some runtime checks in strategic locations so that the proof can be generated can mitigate this problem. On the other hand, the potential of PCC is much larger than SFI, since more than just memory safety can be encoded in proofs.

Typed Assembly Language (TAL) [MCG+99] is a low-level, statically typed target language. The goal of that work is to identify typing abstractions that have general utility for encoding a variety of high-level language constructs and security policies, but that do not interfere with compiler optimizations. This approach may be well suited for producing high-performance, memory-safe native code, when the source files are available.

A more radical approach is to disallow any unsafe native code. For this to be practical, enough compiler and infrastructure support should be provided so that code written in the safe language executes reasonably fast and so that there is no need to resort to native code to access underlying operating system services. The SPIN extensible operating system [BSP+95a] is an example of this approach. Kernel extensions are written in a type safe subset of Modula-3 [Nels91] and installed in the kernel. The extensions can install handlers for any kernel events in which they are interested and for which they have appropriate permissions. Even though the use of a safe language simplifies some of the safety issues, certain aspects of cleaning up after an errant extension are problematic in SPIN. For instance, dealing with resource hoarding is a challenge. While we share the view of the SPIN authors that *protection is a software issue* [BSP+95b], it is important to stress that ensuring memory safety is just one aspect of taming native code. Other aspects, such as conflicting use of OS interfaces, are equally important and as dangerous as errant memory writes.

7.2 Using Address Spaces for Protection

Virtual memory hardware is now ubiquitous, and most of today's operating systems rely on it for ensuring protection, both between the kernel and applications and among applications themselves. Memory protection is enforced by the hardware and is relatively inexpensive.

Most operating systems limit their use of hardware-based protection mechanisms to protection between address spaces. The recent Palladium system [CVP99] is a notable exception. Palladium exploits the Intel x86 virtual memory hardware support for variable-length segments to provide intra-address space memory protection. The basic idea is to put mutually untrusted parts of a program (e.g., a program core and some user-defined extensions of it) into disjoint segments within an address space. A lightweight segment boundary crossing mechanism is provided, and is significantly faster than the best reported IPC mechanisms. However, this approach has several limitations that make it unsuitable for the Java platform. First, extensions must be single threaded. Second, they cannot make arbitrary system calls, which must be mediated by the extended application. Third, it is unclear how extensions can share operating system resources.

Microsoft’s Common Object Model (COM) [Roge97] allows for the same COM components to execute in the same process (in-proc) or in a separate process (out-of-proc). This is accomplished with the help of an interface definition language (IDL), used to describe data passed into and out of out-of-proc components. The reasons behind enabling out-of-proc COM components are very similar to our motivation. The main differences are the level of transparency and automation achieved in our design.

8 CONCLUSIONS

Isolating native code as described in this paper could have significant benefits for JVM development groups and for the users. Several important areas where this is useful include:

- increased debuggability of a single JVM running a single application,
- increased reliability of systems with native libraries which use OS resources in a way conflicting with the use intended by the JVM,
- enabling user-supplied native code in embedded JVMs and in multitasking systems based on the JVM,
- enabling mixed-mode debugging, that is, debugging of an application that contains a mix of Java code and of native code, independent of the JVM, with any Java application debugger and with any C/C++ debugger

(we were able to step through Java code using the Forte™ for Java IDE [Sun00b] and then step through native calls made by the program with gdb; gdb was executing *n-process*),

- using 32-bit libraries with 64-bit JVMs, or vice-versa (we have successfully experimented with running a 32-bit JVM which used a 64-bit native library). The prototype infrastructure is JVM-independent, automated, and portable.

This technique addresses all the identified issues arising when arbitrary native code is used. In addition to offering memory protection, this approach prevents conflict of interfaces and various interferences between the JVMs and native libraries.

The prototype was tested with several implementations of the JVM available on the Solaris Operating Environment. In all cases the system operated properly, without any need to change the runtime. Another important property is that the infrastructure is very easy to use – the names of native libraries to be executed have to be supplied and the rest is being taken care of by a makefile and a set of code-generating scripts. Overall, the functionality, usability, portability and transparency of the infrastructure make it an attractive mechanism for achieving a high degree of control and isolation of native code.

The described design and implementation are heavily influenced by the scope of JNI and as such are specific to the Java programming language. However, the basic approach can be useful to designers of new native interfaces and for those struggling with the very same problems addressed in this paper but in the context of a different programming language.

9 ACKNOWLEDGEMENTS

The authors are grateful for the comments and suggestions made by Dave Dice, Mick Jordan, Hideya Kawahara, Brian Lewis, Tim Lindholm, Bernd Mathiske, Glenn Skinner, Pete Soper, and Dave Ungar.

10 REFERENCES

[AG98] Arnold, K., and Gosling, J. *The Java Programming Language*. Second Edition. Addison-Wesley, 1998.

[BHL00] Back, G, Hsieh, W, and Lepreau, J. *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. Operating Systems: Design and Implementation.* In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA, 2000.

[BG97] Balfanz, D., and Gong, L. *Experience with Secure Multi-Processing in Java*. Technical Report 560-97, Department of Computer Science, Princeton University, September, 1997.

[BSP+95a] Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. *Extensibility, Safety and Performance in the SPIN Operating System*. 15th ACM Symposium on Operating Systems Principles, Copper Mountain, CO, December 1995.

[BSP+95b] Bershad, B., Savage, S., Pardyak, P., Becker, D., Fiuczynski, M., Sirer, E. *Protection is a Software Issue*. 5th Workshop on Hot Topics in Operating Systems, Orcas Island, WA, May 1995.

[BV99] Bryce, C. and Vitek, J. *The JavaSeal Mobile Agent Kernel.* 3rd International Symposium on Mobile Agents, Palm Springs, CA, October 1999.

[CVP99] Chiueh, T., Venkitachalam, G., and Pradhan, P. *Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions.* In Proceedings of 17th ACM Symposium on Operating Systems Principles, Kiawah Island, SC, December 1999.

[Czaj00] Czajkowski, G. *Application Isolation in the Java Virtual Machine.* In Proceedings of ACM OOPSLA'00, Minneapolis, MN, October 2000.

[DBC+00] Dillenberger, W., Bordwekar, R., Clark, C., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M., Samuel, F., and St. John, R. *Building a Java virtual machine for server applications: The JVM on OS/390.* IBM Systems Journal, Vol. 39, No 1, 2000.

[Enge00] Engelschall, R. *Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation*. USENIX Annual Technical Conference, San Diego, CA, June 2000.

[HK93] Hamilton, G., and Kougiouris, P. *The Spring Nucleus: a Microkernel for Objects.* Summer USENIX Conference, Cincinnati, June 1993.

[HCC+98] Hawblitzel, C., Chang, C-C., Czajkowski, G., Hu, D. and von Eicken, T. *Implementing Multiple Protection Domains in Java*. USENIX Annual Conference, New Orleans, LA, June 1998.

[Lang98] Lango, J. *An Implementation of the Solaris Doors API for Linux.* http://www.rampant.org/doors.

[Lian99] Liang, S. *The Java Native Interface.* Addison-Wesley, June 1999.

[LY99] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*. 2nd Edition. Addison-Wesley, 1999.

[MM00] Mauro, J., and McDougall, R. *Solaris Internals: Core Kernel Architecture.* Prentice Hall, 2000.

[Morg98] Morgenthal, J. *A Case for Embedding the JVM into Apps.* InternetWeek, June 22, 1998, Issue 720.

[MCG+99] Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. *TALx86: A Realistic Typed*

Assembly Language. In Proceedings of ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, May 1999.

[Nels91] Nelson, G., ed. *System Programming in Modula-3.* Prentice Hall, 1991.

[NL96] Necula, G., and Lee, P. *Safe Kernel Extensions without RuntimeChecking.* In Proceedings of the Second Symposium on Operating Systems Design and Implementation, Seattle, WA 1996.

[Roge97] Rogerson, D. *Inside COM.* Microsoft Press, 1997.

[SBB+00] Suri, N., Bradshaw, J., Breedy, M., Groth, P., Hill, G., Jeffers, R., and Mitrovich, T. *An Overview of the NOMADS Mobile Agent System.* 2nd International Symposium on Agent Systems and Applications, ASA/MA2000, Zurich, Switzerland, September 2000.

[SES+96] Seltzer, M., Endo, Y., Small, C., and Smith, K. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions.* 2nd Symposium on Operating Systems Design and Implementation, Seattle, WA 1996.

[Skin00]. Skinner, G. *Personal Communication.*

[Solo98] Solomon, D. *Inside Windows NT.* Second Edition. Microsoft Press, 1998.

[Spec98] Standard Performance Evaluation Corporation. *SPEC Java Virtual Machine benchmark suite.* August 1998. http://www.spec.org/osg/jvm98.

[Sun00a] Sun Microsystems, Inc. *The Java Virtual Machine Debugging Interface*. http://java.sun.com/products/jpda.

[Sun00b] Sun Microsystems, Inc. *Forte™ Tools: Forte™ for Java™.* http://www.sun.com/forte/ffj.

[WLA+93] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. *Efficient Software Fault Isolation*. 14th ACM Symposium on Operating Systems Principles, Asheville, NC, December 1993.

[WG94] Weaver, D., and Germond, T. *The Sparc Architecture Manual – Version 9.* Prentice Hall, 1994.

About the Authors

Grzegorz Czajkowski is a Staff Engineer in Sun Labs, and the Principal Investigator for the Barcelona project, which focuses on issues related to safe and scalable execution of multiple applications in a single instance of the Java virtual machine. His interests include the implementation of programming languages, virtual machines, operating systems, and application servers. He holds a Ph.D. in Computer Science from Cornell University.

Laurent Daynes is a Staff Engineer in Sun Labs. His research interests include virtual machine design, language-based extensible operating environment, operating systems, persistent programming languages and advanced transaction processing systems. Before joining Sun Labs in October 1997, he was a research fellow at the University of Glasgow, Scotland, where he was the lead designer and implementer of the first prototype of PJama, a Java virtual machine with a provision of orthogonal persistence. At Sun Labs, he has contributed to the Forest project and is now a member of the Barcelona project. He holds a Ph.D. in Computer Science from the University Pierre & Marie Curie (Jussieu Paris 6), France.

Mario Wolczko is a Senior Staff Engineer in Sun Labs. His interests include the implementation of object-oriented languages, especially the tradeoffs between hardware and software techniques. He has contributed to the Self project, the Exact VM (also known as the Solaris 1.2.2 Production Release), and the KVM. He holds a Ph.D. in Computer Science from the University of Manchester.