A Multi-User Virtual Machine

Grzegorz Czajkowski — Laurent Daynès — Ben Titzer

Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043, USA

S[3] Lab, Purdue University
1398 Computer Sciences Bldg.
West Lafayette, IN 47906, USA

grzegorz.czajkowski@sun.com laurent.daynes@sun.com titzer@purdue.edu

ABSTRACT

Recent efforts aimed at improving the scalability of the Java™ platform have focused primarily on the safe collocation of multiple applications in the virtual machine. This is often beneficial for various performance metrics, but ultimately leads to a *single-user* multitasking environment. The lack of multi-user capabilities forms a barrier to the scalability of multitasking virtual machines, as it requires one per user. In this paper we demonstrate how to enhance a multitasking virtual machine with multi-user support. In particular, users can securely manipulate their private files, load their own native libraries without endangering other computations, and use all standard APIs. Auxiliary processes are needed to provide multiple operating system resource and user contexts, but no modifications are needed to the operating system itself.

1 INTRODUCTION

Program execution environments based on safe languages have become an important part of the computing landscape, as demonstrated by a growing number of middleware systems taking advantage of the Java platform [GJS+00]. The scalability of the underlying virtual machines is key to efficient resource utilization and consequently to widespread acceptance of safe languages. Several recent projects have demonstrated that scalability can be improved by re-architecting the run-time system or by program transformations that enable execution of multiple applications in a single instance of the virtual machine with certain degrees of application isolation [HCC+98, BV99,BHL00,CD01]. Although the results of these efforts differ considerably with respect to features available and performance, they are invariably multitasking *single-user* environments.

The single-user behavior manifests itself in several ways. First, since the run-time system executes as a single operating system (OS) process, with a single set of user privileges and session attributes, private files of only the user whose effective user id the virtual machine process has can be accessed. Typically 'running as root' to address this problem is not an option, as it is dangerous from the security viewpoint. Setting the effective user id affects the whole process, which would only be correct if all file access operations performed by the virtual machine were serialized. The second issue concerns user-supplied native libraries. Most multitasking systems based on safe languages do not allow such code, since a malfunction might jeopardize the whole environment and all the applications in it. Finally, the correct execution of certain core components of the safe language platform, such as the windowing subsystem, is not guaranteed in presence of multiple users, or even when a single user runs multiple applications that require such components. The issue here is the interference of unrelated computations via global state of core native libraries.

Our previous work has demonstrated that collocating of computations in a multitasking virtual machine combined with aggressive sharing of run-time data structures can improve performance and significantly decrease start-up time and memory footprint [CD01]. To fully realize its potential, this approach must address the multi-user issues mentioned above. A case in point is thin-client environments where stateless desktop consoles access a shared pool of computational resources in one or more powerful servers [SLN99]. At peak times the number of active users on a single Sun Ray™ installation can reach hundreds. If every user runs just a single application in a dedicated Java virtual machine (JVM™), the combined resource requirements severely stress the system and negatively impact the user experience. The bottom line is that a single multi-user multitasking virtual machine offers the potential to utilize resources better than a collection of single-user multitasking virtual machines, just as a single-user multitasking virtual machine scales better than a collection of virtual machines each executing a single application.

This paper demonstrates that it is possible to construct a complete and fully compliant multitasking multi-user virtual machine for executing programs written in the Java™ programming language. Our solutions take advantage of the existing OS infrastructure and do not require high engineering effort. In particular, we

enhance the Multitasking Virtual Machine (or MVM) [CD01] with the ability to execute applications of different users. This includes correct maintaining of user identity, including access control of users' private files, the ability to run user-supplied native code safely, and a mechanism to ensure correct operation of core native libraries in the presence of multiple users. The last feature is based on a novel technique that transparently replicates the global state of shared libraries. The first two enhancements take advantage of the ability to pass open file descriptors among processes and improve on MVM's ability to isolate native code.

The rest of the paper is structured as follows. Section 2 contains an overview of the architecture, Sections 3-5 describe the handling of user identity, user-supplied native code, and core native libraries, respectively, along with performance details. A discussion of design alternatives and related work is given in Section 6.

2 ARCHITECTURE OVERVIEW

The proposed multi-user virtual machine architecture, dubbed MVM-2, is described later on, after an introduction to MVM (this term will consistently refer to the previous version of the system [CD01]).

2.1 Background on MVM

MVM is a general-purpose virtual machine for executing multiple applications, written in the Java programming language, on behalf of a single user. It is based on the Java HotSpot™ virtual machine (referred to from now on as HSVM) [Sun00a] and its client compiler, version 1.3.1 for the Solaris™ Operating Environment [MM01]. In experiments described in the following three sections version 2.9 of the operating environment was used, running on a Sun Enterprise™ 3500 server with four UltraSPARC™ II processors and 4GB of main memory.

Applications executing in MVM are referred to as *isolates* [JCP01]. MVM-aware applications, such as application server engines, can use the provided API to control the life-cycle (e.g., creation and asynchronous termination) of other isolates. The main (first) isolate does not have to be a server – it can be any application written in the Java programming language. A simple example of the API is the creation of an isolate, which will execute MyClass with a single argument "abc":

```
new Isolate("MyClass", new String[] {"abc"}).start(...);
```

The key design principle of MVM was to examine each component of the JVM and determine whether sharing it among isolates can lead to any interference among them. Non-shareable components are either replicated on a per-isolate basis or made *isolate re-entrant*, that is, usable by many isolates without causing any inter-isolate interference. They include static fields, class initialization state, and instances of java.lang.Class.

Shareable components that require modification to become isolate re-entrant include the constant pool, the interpreter, the dynamic compiler, and the code it produces. An arbitrary number of isolates in MVM can share the code (bytecode and compiled) of both core and application classes. Runtime modifications make the replication of non-shareable components transparent. In effect, each application "believes" it executes in its own private JVM, as there is no interference due to mutable run-time data structures visible directly or indirectly by the application code. Similarly, certain Java Development Kit (JDK™) classes, such as System and Runtime had to be modified to make operations such as System.exit() apply only to the calling isolate.

The heaps of isolates are logically disjoint. The separation of isolates' data sets in MVM implies that isolates cannot directly share objects, and the only way for isolates to communicate is to use copying communication mechanisms, either standard ones, such as sockets, or low-level custom protocols [PCD+02]. Another option is to use *links*, which are a low-level isolate-to-isolate communication mechanism introduced by the isolate API [JCP01].

In MVM most of the class representation is shared, and so is the class loading, linking, and run-time compilation effort. In particular, only when a class is loaded into MVM for the first time the actual file fetching, parsing, verification, building of a main-memory run-time representation of the class, and several other steps are performed. They do not need to be repeated when another isolate uses the same class.

2.2 Process Model

Essential to bringing multi-user capabilities to MVM is a process model that encapsulates the ideas of protection and access control. In our design, a single instance of MVM-2 exists as one process. It contains multiple isolates. Isolates may be started within the JVM by different users through a separate login program called *Jlogin*, written in C. Jlogin corresponds to a notion of a user session, and is used to start a single isolate – the user simply types in the name of the main class and its arguments, similarly to running the standard "java" command.

After session initialization, Jlogin serves as a daemon process that services certain requests issued by the initial isolate in the session; these requests are related to (i) user identity – this breaks down into several sub-cases: accessing the file system, accessing environment information related to the user session (i.e., an instance of Jlogin), and maintaining user

identity and session attributes across process creation via the Runtime.exec() method, and (ii) interacting with user-supplied native code. The Jlogin process has the effective user id and associate privileges of the actual user, regardless of process attributes of MVM-2 (Figure 1).

The initial isolate has its standard input/output/error streams connected appropriately to its Jlogin process; these streams may be shared with descendent isolates (isolates created in the course of the program execution, and not by spawning another Jlogin process) if the initial one sets them so. Each isolate, regardless of how it was created, has its own instance of Jlogin (lazily created for non-initial isolates), so that a failure of native code associated with one isolate does not affect the others. Optimizations to this basic scheme, such as optional association of multiple isolates with a single Jlogin process, are not further pursued in this paper.

Each user can have multiple sessions. To simplify further discussion let us assume that the initial process does not spawn further isolates and that there is only one Jlogin per user.

The first isolate of MVM-2 is a simple application called Mserver that listens on a socket for connections from Jlogin processes. Each new Jlogin connects to Mserver and the two exchange information such as relevant environment variables and user settings. Jlogin then sends a request to Mserver to create an isolate to run the application the user specified when starting that Jlogin instance. The isolate connects to its Jlogin's standard input, output, and error streams.

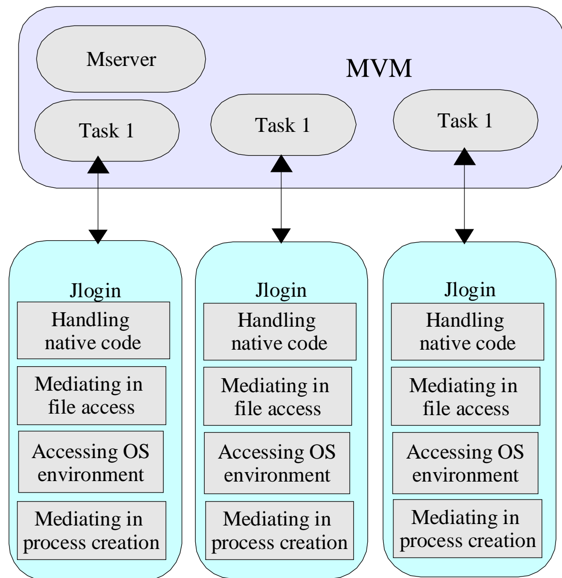

Figure 1. Three users execute applications in MVM-2; each of them has an instance of the Jlogin process.

Multiple Jlogin processes from different users can connect to the Mserver to launch their applications within the same instance of MVM-2.

3 USER IDENTITY

In MVM-2 the user identity and session attributes of Jlogin are associated with isolates. In particular, users are able to access their private files securely, and at the same time are not able to elevate their own privilege or circumvent the OS access control mechanisms. Moreover, user identity is properly preserved across process creation. These issues are discussed below.

3.1 File Access

A straightforward approach to enable secure file access for multiple users might be to modify the JVM so that it performs explicit access control checks for file operations that are at least as restrictive as the underlying OS. Upon a user request, the JVM would first decide whether a file system operation was permitted for that user and if so, perform that operation. This scheme would work successfully if the JVM itself had the right to perform all the operations requested by all users – if it was running as root on a UNIX(R) system, for example. However, running the JVM itself with the highest privilege could damage the system if the JVM itself crashes or is compromised. Moreover, properly emulating the underlying system's access rights checks seems quite complex.

To address these issues we have designed the Remote File Access (RFA) subsystem of MVM-2. RFA forwards certain file system operations, such as file opening or deleting, to the Jlogin process associated with the requesting isolate. To accomplish this, open file descriptors must be passed between the MVM-2 and Jlogin processes. This feature is available in several UNIX inter-process communication (IPC) mechanisms, such as unnamed stream pipes, UNIX domain sockets, or streams [Stev90]. Our implementation of RFA uses doors [MM01], a high-performance IPC mechanism available on the Solaris Operating Environment. Operations such as read, write, seek, and close need not be done remotely, since they are just operations on opened file descriptors local to the virtual machine and are not subject to access control checks. Thus such performance-critical operations as reading, writing, and seeking do not incur the cost of IPC. Doors allow for examining clients' credentials. In particular, the process id of the caller can be obtained. This information is used by Jlogin to verify that RFA requests are issued by the given instance of MVM-2.

As Jlogin runs with the access privileges of the user who started the isolate, the OS access control mechanism for this process enforces the correct privilege level for the corresponding isolate. In this

way, no elevation of privilege occurs and access to the user's private files is permitted correctly, in accordance with the UNIX access control semantics. It is not required that MVM-2 run at the highest privilege level. It can be started by any non-privileged user. The Java security model is unmodified – appropriate permissions (i.e., instances of java.io.FilePermission) are still a necessary prerequisite to file access.

As MVM-2 is JDK 1.3.1-compliant, it does not have the New I/O (NIO) APIs [Sun02], introduced in JDK 1.4. One of the classes defined there, java.nio.channels.FileChannel, allows application code to create file locks held on behalf of the entire JVM (e.g., implemented through the fcntl() system call). This is a potential interference point for multiple computations collocated in the same instance of the JVM. Similarly to how MVM-2 gives each isolate secure access to all of the corresponding user's files, a 1.4-compliant MVM-2 would forward file locking operations to Jlogin to provide locking semantics indistinguishable from a model in which each application executes in a dedicated virtual machine process.

The commands implemented in the RFA protocol are the following: RFA_open (open a file given a pathname), RFA_mkdir (create a directory), RFA_mode (get the access mode bits), RFA_getmtime (get the modified time of a pathname), RFA_length (get the length of a file), RFA_access (check access to a given pathname), RFA_list (return a list of files in a given directory), RFA_setmtime (change the modified time of a file), RFA_remove (remove a given pathname), RFA_protect (write-protect a given pathname), and RFA_rename (rename a given pathname). These commands form a basic set of operations needed to implement the semantics of the java.io classes that operate on files.

3.2 Process Creation

Another issue related to user identity is creating new processes. The Java programming language allows applications to execute arbitrary programs as new processes via the exec() method of the java.lang.Runtime class, and to kill or wait for these processes via the methods of the java.lang.Process class. Standard implementations of the JVM utilize the fork/exec capabilities of the underlying OS to provide this functionality.

MVM-2 ensures that a process started by an application running on behalf of a particular user inherits not just the privilege of that user, but also the environment of the process that launched that application. Running MVM-2 with root privileges and then "fork-exec-ing" it combined with using setuid() would not adequately address this issue, as the environment attributes would be of the MVM-2 process and not of the appropriate Jlogin process. Our solution is similar to the way MVM-2 deals with file accesses: requests to spawn a new process, to wait for its completion, or to kill it are forwarded to Jlogin of the current isolate. This guarantees that the new processes runs with both the correct user identity and inherits the appropriate environment properties (e.g., current directory, environment variables, etc.).

3.3 Accessing Environment Properties

Even though the Java platform does not provide an API to perform an equivalent of getenv() and setenv() available on the UNIX platforms, internally the JDK accesses the environment, for example to obtain the values of the TZ (time zone) and DISPLAY variables. MVM-2 forwards these requests to Jlogin.

3.4 Changes to the JVM and the JDK

RFA required only one minor change to HSVM: the file opening operation had to be modified to select the correct Jlogin process to forward the RFA request to. This operation, used internally by the native code of some core classes, is encapsulated in the internal JVM file opening call, which ultimately calls the OS.

Changes were required to classes in the java.io package. The non-public abstract class FileSystem encapsulates the functionality of the file system for other classes in the package. It has a static method used to retrieve an object that represents the correct underlying file system, for example an instance of UnixFileSystem. The only change to FileSystem was to make this static method return an instance of RFAFileSystem instead. RFAFileSystem extends UnixFileSystem making remote RFA calls to forward requests to the appropriate instance of Jlogin.

Supporting the required behavior of Runtime.exec() and the Process class requires modifications of the native methods of the sub-class of Process that implements process support for a particular OS. The modifications consist of forwarding the request for fork()/exec(), wait(), and kill() to the Jlogin process. Input, output, and error streams are properly set for the new process. Finally, minor changes were required for forwarding queries about environment attributes from MVM-2 to Jlogin.

3.5 Performance

There is no performance impact on file operations that do not require an IPC to Jlogin, such as reads and writes. Also, socket operations do not suffer any of the IPC overhead, since the original JDK java.net code did not have to be modified for MVM-2.

Table 1 summarizes the overheads of RFA on java.io.File operations that need to be mediated by Jlogin. The additional cost is between 81% for mkdir() and 268% for lastModified(). The variance in relative

Operation	Overhead
open()	261%
length()	257%
lastModified()	268%
renameTo()	125%
setReadOnly()	146%
isFile()	178%
canWrite()	242%
list()	141%
mkdir()	81%

Table 1. RFA overheads on java.io.File operations.

overheads are explained by different costs of these operations when they are not forwarded to Jlogin: more expensive operations such as mkdir() and list() incur relatively lower overhead than the cheaper ones, such as length(). Moreover, open() incurs an additional cost of passing an open file descriptor via a door call, which is more expensive than "plain" door calls. Even though the other operations do not need to pass open file descriptors between Jlogin and MVM-2, they still need to be forwarded to Jlogin. Otherwise, insufficient level of privilege may prevent, for example, listing of a directory.

The impact of these operations on the actual file manipulation depends on how many of them are issued relative to the number of reads and writes. For instance, opening a file (remote operation) followed by a hundred 80-byte FileOutputStream.write(byte[]) operations and a close (all local operations) is 27% more expensive than when run with an unmodified HSVM, while the same sequence with a thousand writes is 2.8% more expensive. Another micro-example can be the creation of a properties object from a file:

```
new Properties().load(new FileInputStream(fileName));
```

The overheads depend on the size of the named file. For the *flavormap.properties* distributed with the JDK 1.3.1 (size: 929 bytes) the overhead when compared to HSVM is 18%. For *psfontj2d.properties* (size: 10669 bytes) the overhead is 7.7%.

How this translates into application execution time overheads depends on the intensity of using RFA-mediated file operations. For example, the performance impact of RFA on the file system intensive *javac* benchmark from the SpecJVM98 benchmark suite [Spec98], is less than 0.5%.

4 USER-SUPPLIED NATIVE CODE

The coexistence of programs written in a safe language with user-supplied, unsafe (native) code is convenient, as it enables direct access to hardware, OS resources, and legacy code and can improve performance. But the inherent lack of memory safety in native code may break the contract offered by a safe language. In the case of a single application executing in the JVM, a bug in an application (user-level) native library will disrupt or abnormally terminate this particular application only. The consequence of an errant native library carelessly loaded into MVM-2 can be much more serious. In addition to causing the loading application to malfunction, such a library may corrupt the data of other applications, perform arbitrary operations with the privilege level of the virtual machine, or crash the whole virtual machine, causing denial of service.

Memory safety is not the only issue, though. Guaranteeing the conflict-free use of system resources by the JVM and native code is equally important. Native code is written against two interfaces: the Java Native Interface (JNI) [Lian99], which is the sole interaction point between the JVM and the application, and the host OS interfaces, involving the usual libraries for I/O, threading, math, networking, etc. The latter is also the interface against which the JVM is written. The problem is that the JVM makes certain decisions regarding the use of the host OS interface and of available resources, and these decisions may conflict with their use by the user-supplied native code. For example, signal handlers may have to be instantiated to handle exceptions that are part of the operation of the JVM (e.g., to detect memory access and arithmetic exceptions). Another example is the JVM's choice of a memory management regime for purposes such as the allocation of thread stacks. In each of these cases an arbitrary use of any of these resources by user-supplied native code can cause the virtual machine to malfunction.

MVM dealt with these issues by automatically and transparently executing user-supplied native libraries in a separate process. Each isolate that needs it and has the necessary permissions has one such process. This means that the only interface between the JVM and native libraries becomes JNI. There is then no implicit contract concerning memory management, threading, signal handling, and other issues. This refactoring solves the composability problem neatly. The native code in a separate process has full control of its own resources. There are no unexpected interactions with MVM via memory, signals, threads, and so on. However, the design of MVM's native code isolation was not suitable for multiple users. The problems are described in Sec. 4.2, which is followed by the description of the new design. First, the essential information on JNI is given.

4.1 JNI Essentials

JNI interacts with the JVM via *downcalls* (when a Java method calls a native method) and *upcalls* (when a native method requests a service from the JVM). Upcalls enable accessing static and instance fields and array elements, invoking methods, entering and exiting monitors, creating new objects, using reflection, and throwing and catching exceptions.

Downcalls result in calls to C or C++ functions, whose names are generated by the *javah* tool from the names of methods declared as native. The naming convention is Java_packageName_className_methodName. An optional signature may also be appended to the end of the name to support C++ or to disambiguate overloaded method names. The JVM uses this naming convention to bind the address of an exported function to that of the native method at invocation time.

Upcalls are invoked via a *JNI environment* interface, a pointer to which is always passed as the first argument to all JNI upcalls and downcalls. Objects, classes, fields, and methods are never accessed directly, but rather via appropriate opaque references or identifiers. These references are meaningful only to the JNI functions, and shield native code from the details of particular implementations of JNI.

4.2 Previous Native Code Isolation

Native code isolation (NCI) is achieved in MVM by generating ahead of time a proxy library for each specified library holding native methods. For each function in the original library there is a function with the same name in the corresponding proxy library. The JVM's environment variable LD_LIBRARY_PATH is manipulated such that the virtual machine will load proxy libraries with the same name; the original (unmodified) libraries are loaded into a remote NCI process. From that point on, the proxy library and the NCI process orchestrate, transparently to the original native code, the forwarding of native method invocations and JNI upcalls across process boundaries.

The functions in proxy libraries are redefined to forward their arguments, along with information uniquely identifying the function, to a dedicated, transparently created process. Upon receipt of such a request, the process executes the required function with the supplied arguments. Just before the execution of the function, the first received argument is replaced with a custom JNI environment pointer. This custom JNI environment redefines all JNI upcalls so that each of them ships all of its arguments along with its unique identifier back to MVM, where the upcall is dispatched to the JVM's actual implementation of the JNI call. Upcalls are always executed in the same thread that issued the original downcall. For instance, an exception thrown in an upcall has to be dispatched to the thread that caused the downcall.

However, the proxy library approach proved to be inflexible for MVM-2 because: (i) it requires manual intervention to generate the proxy libraries before executing the application, (ii) loading multiple libraries in the same remote process is not dynamic, (iii) allocation of method identifiers is static and not unique across the JVM, (iv) the proxy library would have to be redesigned to dispatch to different instances of Jlogin based on an isolate identifier.

4.3 New Design

As in previous design, in MVM-2 user-supplied native libraries are loaded into a separate process – in our current implementation Jlogin serves this purpose. All invocations of native methods and of JNI upcalls are executed remotely via door calls transparently to both Java methods and user-supplied native code. If an error occurs during the native method invocation, for instance if the Jlogin process aborts due to a bus error caused by a user-supplied native library, MVM-2 throws a Java exception in the invoking thread. The exception is unchecked so that existing code is not broken, but still can be caught by applications coded to deal with unexpected failures. Just as in the proxy approach, a custom JNI environment pointer is used to ensure proper forwarding of upcalls back to MVM-2.

Even though in MVM-2 multiple isolates transparently share class and method representations, different isolates from different users are unaffected by the behavior and bindings of each other's native methods. This includes correct handling of pathological cases such as different isolates resolving the same method of the same class to different native methods in different native libraries. To this end two identifiers are introduced: a global (JVM-unique) method id and an isolate-unique library id.

A JVM-unique method id is assigned to each native method at class load time. This id is used in the Jlogin process to bind a native method id to the actual native code that should be called. The binding is performed in Jlogin which guarantees that a particular isolate's bindings do not affect the bindings of any other isolate.

An isolate-unique id is assigned to each loaded library (during System.loadLibrary()), identifying a particular library in an isolate's Jlogin for the purposes of method resolution and unloading. This assignment is done at library load time by Jlogin, and the id is later used only between an isolate and its Jlogin.

These two unique identifiers are sufficient to support library loading and unloading as well as method resolution and invocation with the required semantics of giving each isolate an illusion of being the only one in the entire virtual machine.

4.4 Changes to the JVM

Some modifications to the JVM itself were necessary in order to accomplish the goals of the new NCI protocol. MVM was modified to track per-isolate information needed to locate Jlogin for each isolate and to forward load and unload requests to the correct

remote Jlogin process. Similarly, the native method dispatching mechanism of the JVM was modified to forward the native call to the correct Jlogin process.

Library loading modifications included decisions as to whether a native library should be loaded locally within the JVM itself and which libraries to load in the Jlogin processes. Libraries loaded by core classes should not be isolated, and native libraries needed for the actual implementation of NCI must not be isolated. In MVM-2 both of these cases are handled in the same way and are loaded into the virtual machine's process.

On the other hand, user-supplied native libraries must be isolated. In our implementation of the JDK, instances of ClassLoader track the currently native loaded libraries for each class. This required modifications to determine which libraries to load remotely and which libraries are "trusted" system code. Finally, the HSVM code that builds run-time representations of methods from class files has been modified to assign JVM-unique method ids to native methods of loaded classes.

4.5 Performance

Much like the overheads of RFA, the performance impact of NCI depends on how often it is used. Isolates that do not invoke user-supplied native methods do not incur any performance penalties. Programs frequently issuing JNI calls may suffer significant overheads. For instance, an empty static downcall through NCI is about 588 times slower than the same call in an unmodified JVM (in absolute terms, the difference is between tens of microseconds versus tens of nanoseconds). Upcalls incur smaller yet significant penalties – a static method upcall is about 45 times slower through NCI than in HSVM. The difference between the NCI's downcall and upcall overheads when compared to HSVM is explained by the fact that in HSVM upcalls are much more expensive than downcalls, and MVM-2 introduces the same overhead to both downcalls and upcalls. However, a vast majority of programs do not issue such a volume of JNI downcalls and/or upcalls related to non-core native libraries to make performance degradation due to this technique noticeable.

5 VIRTUALIZING CORE LIBRARIES

Core native libraries contain the implementations of native methods from core packages, distributed with the JDK. These libraries were typically coded without much thought directed toward safe in-JVM multitasking, let alone multiple users, and contain a substantial amount of static (global) state. To ensure isolation, each isolate needs to have its own copy of such state.

It would seem desirable to handle core native libraries in the same way user-supplied native libraries are executed in MVM-2 – in a separate process. This would provide each isolate with its own copy of core native libraries, and in particular with its own instance of their data segments. We experimented with this approach but soon discovered that certain components of the Java platform make extensive use of their native libraries, and the traffic across the JNI boundary may be heavy. Performance overheads made the use of this technique particularly unattractive for executing native code of core classes associated with the Abstract Window Toolkit (AWT). For example, during the start-up of the Notepad demo application distributed with the JDK there are 2592 downcalls and 156 upcalls. NCI overheads (Sec. 4.4) increase start-up time by an order of magnitude.

Core native libraries are as trusted and robust as the virtual machine itself, and are under full control (i.e., can be modified) of the JVM developers. This observation suggested that only a single instance of each of them may need to be loaded into the virtual machine, provided that the libraries are modified in two ways: their static state is (manually) replicated, and any interaction with the underlying OS is examined to guarantee the absence of inter-isolate interference. This approach still was not satisfactory, since depending on the JDK's implementation the amount of changes required to completely analyze and modify the core native libraries may be substantial. Moreover, the modifications would not extend to the X Window libraries, which are distributed with the operating system and not with the JDK; AWT native code depends on these libraries. GUI-enabled isolates would thus still interfere through the static state of, for example, libX11.

The solution we eventually adopted has none of these performance and engineering disadvantages. It is based on a technique that allows for loading multiple instances of the same dynamic library into a single process. In particular, the advantages are (i) memory footprint related to core native libraries does not increase relative to HSVM, and (ii) only minor modifications to the JDK (about 20 lines of code) were required to apply this technique to the AWT subsystem to provide per-isolate static native state. The applicability to Swing/AWT is particularly interesting, as these JDK components generate large amounts of meta-data (loaded classes, compiled method, etc.). Since meta-data is shared in MVM-2, the ability to execute GUI-enabled isolates increases the scope of memory footprint savings and at the same time applies MVM-2's start-up time reduction to interactive applications.

5.1 Basic Technique

The Solaris Operating Environment supports a *runtime linker auditing interface*. Audit libraries can monitor and modify certain operations, such as mapping of shared binary objects into memory and the binding of symbols in a customized way [Sun00b]. Note that we use the term *shared binary object* to refer to a concatenation of relocatable objects (.o files) that provides services that might be bound to a dynamic executable at runtime. A shared binary object may have dependencies on other shared object. In the linker/loader literature the typically used term is *shared object*, which may be confusing when used along the concepts from object-oriented programming.

In order to isolate audit libraries from the symbol binding requirements of the audited applications and shared binary objects, the interface employs its own *link-map* list, with each entry on the list describing a single shared binary object within the process. The symbol search mechanism required to bind shared binary objects for an application traverses this list of link-maps. Thus, the link-map list provides the name-space for process symbol resolution.

The runtime linker itself is also described by a link-map, which is maintained on a different list from that of the application objects. Having the linker reside in its own unique name space prevents any direct binding of the application to services within the linker. The audit interface allows for the creation of an additional link-map list, so that audit libraries are also isolated from the symbol binding requirements of the application.

We have taken advantage of this infrastructure to load multiple copies of a library into a single OS process. Each such library is loaded by the dlmopen() function [Sun00b] on a new link-map list and at a virtual address different from any other instance of itself in the same process. However, text segments of all instances of the library, regardless of what process's virtual memory they are mapped into, are backed by the same physical memory pages.

5.2 Application to AWT

The technique described above has been applied to MVM-2 and AWT-related libraries. Arguably out of all JDK components with native libraries AWT is the most complex one. For instance, it starts its own threads to handle events and depends on X11 and other related libraries (e.g., Motif), which are in themselves quite complex.

In a nutshell, the approach is to group together the entire set of AWT-related shared libraries as well as the libraries they depend on (such as libX11, etc.) into a unit, called from now on *the AWT context* (or simply *context*). All libraries in the same contex share the same unique link-map list. Due to the name-space isolation provided by separate link-map lists each AWT context can be loaded multiple times within the same OS process without the danger of interference with other contexts. The above is insufficient, however, to provide each isolate with independent AWT capabilities. The major issues that needed to be addressed to make this scheme work are: (i) managing the interface between the virtual machine and multiple AWT contexts, (ii) handling the dependencies of the libraries in AWT contexts on the virtual machine, and (iii) preventing conflicting use of OS resources by multiple AWT contexts.

5.2.1 The JVM-core native libraries interface

Maintaining one AWT context per isolate requires dispatching each invocation of an AWT native method to the AWT context associated with the current isolate.

The JVM (and MVM-2) interface with native code via JNI, and thus the names of core native methods conform to the same stylized naming scheme as an ordinary native library. A simple script based on the standard *nm* utility (listing the symbol tables of shared binary objects) and on *javap* (disassembling classes) is sufficient to generate a list of all such methods, along with their signatures, from the libraries comprising the AWT context. The list is then used to generate *libawt_proxy.so*. At boot time MVM-2 loads a single instance of this library in the main context (i.e., JVM's context). Each function defined there forwards its invocation to an appropriate per-isolate instance of an automatically generated *libawt_context.so*. A new instance of this library is loaded by libawt_proxy.so, using dlmopen(), whenever a new isolate is started. The library is a part of the AWT context, and contains all of the AWT-related JDK and X11 libraries in its list of shared binary object dependencies. Thus loading an instance of libawt_context.so loads a new set of instances of these libraries as well, that is, the entire context.

Invocation forwarding does not require any changes to the JDK or the runtime system. Whenever a native method is called, the runtime system finds the required name in libawt_proxy.so and calls it (Fig. 2, left side). Only there the actual look-up of the isolate identifier and the associated AWT context takes place. For example here is a method from libawt_proxy.so:

```
void Java_java_awt_Color_initIDs(JNIEnv *env, jclass cls) {
   int iid = get_isolate_id();
   context ctx = contexts[iid];
   (*ctx).Java_java_awt_Color_initIDs(env, cls);
}
```

5.2.2 Dependencies on the JVM

AWT native methods invoke Java methods (both application callbacks and services offered by the JDK libraries) exactly as any other native code does: by

invoking the JNI functions of the JNI environment a reference to which is always a first argument to any native method. Although these JNI upcalls are defined in the library that implements the JVM (libjvm.so), they are exported to native methods as function pointers. Due to this, JNI functions do not create a dynamic linker dependence from AWT native libraries on libjvm.so.

This is however not the case with JNI utility (JNU) functions, which are also defined in libjvm.so. Introduced for convenience, each JNU function groups a common sequence of JNI upcalls. An example of a JNU function is invoking a static Java method by its (string) name, class name, and signature. Accomplishing this with JNI requires six upcalls, including appropriate error checking. In contrast to plain JNI upcalls, JNU functions are called directly (i.e., without a function pointer indirection) by AWT native methods, which makes AWT native libraries depend on libjvm.so. Such dependencies are undesirable, as loading an instance of libawt_context.so would also cause the loading of a new instance of the JVM's libraries along with the new AWT context. In the best case this would only waste memory; in the worst, it can lead to a process crash if a conflict occurs among multiple JVM contexts. For example, each of them may zero out the heap during initialization.

In MVM-2 such dependencies are prevented by avoiding direct references to the JNU functions. Instead, function pointers are used, much like in the case of the JNI upcalls. In order to achieve this while avoiding the modifications of the original AWT libraries, a new shared library, libjnu+system.so, is interposed between the libraries of an AWT context and the JVM. The new library defines all the JNU functions that are used by AWT contexts, and stores the addresses of their original (i.e., defined in the loaded instance of libjvm.so) implementations in a vector of addresses vm_context. Each such interposing function simply consists of calling the corresponding JNU function of the JVM via the function pointer listed in vm_context. The vector is passed to an initialization routine in libjnu+system.so when a new instance of libawt_context.so is loaded.

5.2.3 System calls

Loading multiple AWT contexts can cause inter-isolate interference because the contexts share the system call interface. For example, while calls to getpid() from different contexts are not dangerous, the same cannot be said about sbrk(). Each context's sbrk() would initialize the amount and initial address of space allocated for the process's data segment to the *same* value. Subsequent memory allocations through for example malloc() invoked from different contexts would return the same address, leading to memory corruption. It is thus vital to neutralize all potentially conflicting uses of system calls.

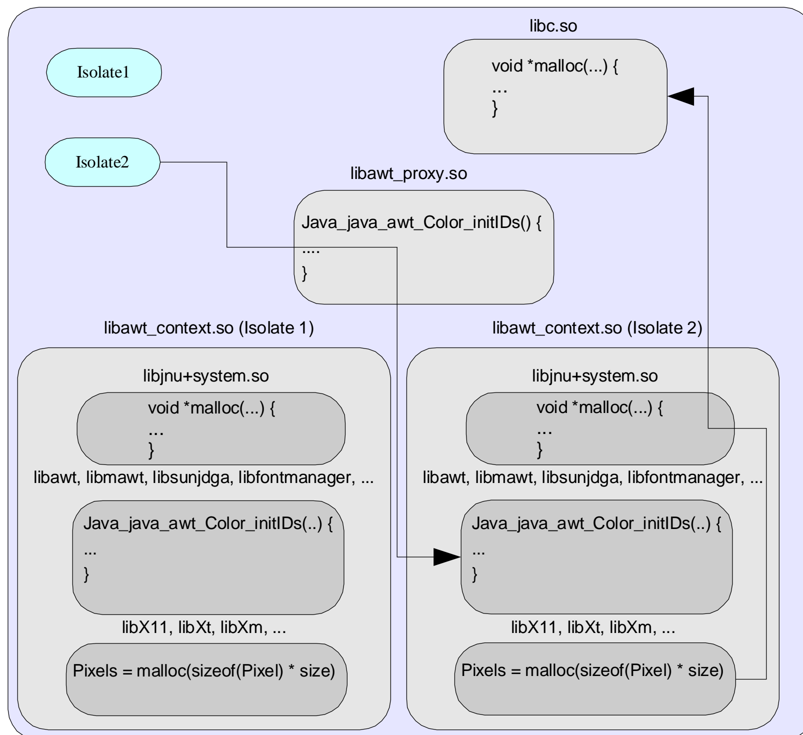

Figure 2. Forwarding of AWT native method invocations to the appropriate AWT context (left) and forwarding system and library calls from an AWT context back to the virtual machine's context.

This issue is solved by extending the technique described in Sec. 5.2.2, where a vector of JNU functions is passed down to an AWT context so that their invocations are properly forwarded back to the virtual machine's context. For this purpose vm_context is extended with addresses of (i) system functions (e.g., sbrk()), (ii) derivative library functions (e.g., malloc()), and (iii) other library functions the use of which must be confined to the virtual machine's context (e.g., dlmopen()). By itself this does not guarantee that the OS interface is used in an interference-free way, but at least introduces a point of programmable control over potentially dangerous behavior. For example, malloc() and free() behave as

expected in this scheme, while the usage of signal() system call may have to be modified, for example by injecting transparent chaining of signal handlers and ensuring the chaining will be respected by all contexts. In MVM-2 this has not been necessary for the AWT subsystem. Figure 2 (right side) shows the forwarding of system calls and virtual machine utility functions.

Another way to approach the forwarding of system calls to the main context is to take advantage of the runtime linker auditing interface's ability to intercept requests for loading a new library. This way, a request to load a context-private instance of libc.so could return a library with the necessary redirections, similar to libjnu+system.so. This technique avoids loading an instance of libc.so along with each context. In MVM-2 a new AWT context contains an instance of libc.so, which is never used. As the segments of libc.so are mostly read-only, this does not increase the overall memory footprint, but unnecessarily wastes virtual memory, which may be an issue for 32-bit virtual machines.

5.2.4 JDK modifications

No changes to the virtual machine were needed to enable the replication of the AWT subsystem. Several minor JDK modifications were necessary, though.

In the original AWT native libraries XOpenDisplay() is invoked with a null argument. This means that X11 will call getenv("DISPLAY") to obtain the value of the DISPLAY variable defined in the environment of the virtual machine. MVM-2 obtains this value by contacting the Jlogin process and then passes it to XOpenDisplay() so that each user session has its own value of DISPLAY.

Other modifications concern JNI_OnLoad. Several native libraries that comprise the AWT context define this optional function, which is invoked during the loading of the library to properly initialize it. In MVM-2 the names of all these occurrences of JNI_OnLoad are changed to <libname>_JNI_OnLoad to avoid name clashes. Then, libawt_proxy.so invokes all of these methods in the original order when loading a new instance of libawt_context.so.

5.2.5 Performance and start-up time

Execution time overhead of the presented technique is negligible (we were unable to detect any during our measurements). The reason is that MVM-2 adds only a few machine instructions to JNI downcalls, and even fewer for JNU upcalls and system calls. MVM-2 enjoys the same performance improvements as MVM, since the effort necessary to load classes and compile methods is saved due to meta-data sharing.

In MVM-2 start-up time of GUI applications improves dramatically relative to HSVM. Table 2 summarizes the results for two demo programs distributed with the JDK: Notepad and SwingSet2. The first one is generally deemed more representative of desktop applications. The second one creates a large number of Java objects during its start-up, and thus the improvements are relatively smaller, although still very much noticeable by the users. Let us note that we have used the following definition of start-up time for AWT applications: it is the time elapsed between invoking the main method of the application and draining of the AWT event queues. The "first" column reports the time, relative to HSVM, necessary to start up the first instance of the application in MVM-2. The "second" column reports start-up time of the second instance (relative to HSVM's start-up time, which is the same for any instance of the benchmark, as it includes starting a new process, JVM bootstrap, etc.) Start-up times of subsequent (third and later) instances are similar to the start-up time of the second instance. This latter number is more typical of the actual user experience, as most of the start-up time decrease is attributed to having already loaded Swing and AWT classes.

	First	**Second**
Notepad	99.3%	28.7%
SwingSet2	99.5%	37.6%

Table 2. Start-up time in MVM-2, relative to HSVM.

Because of the adopted start-up time definition, bootstrap of the virtual machine is not included in the measurements. If it was, the gains would be even higher, as in MVM the time required for preparation of an isolate to run application code, including running static initializers of bootstrap classes, is only 4% of the time required to boot HSVM. Still, the improvements remove between 62-71% of the start-up time overhead, which translates into shaving off hundreds to thousands of milliseconds.

The barely measurable improvement for the first instance is due to the fact that some classes normally loaded in HSVM during the start-up Notepad and SwingSet2 have already been loaded in MVM-2 by the Mserver isolate manager (Sec. 2.2) before the benchmarks are executed.

Any improvements to the JDK classes will have a direct impact on start-up time in MVM-2, as all static initializers of classes needed by an application are run in the same order in an isolate as they are in HSVM.

5.2.6 Memory footprint

There are several components of the memory footprint of an application executing in MVM-2: (i) Java objects created on the heap, (ii) space occupied by the user-supplied native libraries, (iii) meta-data, such as bytecodes, constant pools, and compiled methods, and (iv) space occupied by core native libraries. The

amount of memory used by the first two is the same in HSVM and in MVM-2.

The size of memory required by native libraries related to AWT in MVM-2 is summarized in Table 3. The size of libawt_proxy.so is 576KB, most of which is read-only. This library is not a part of the AWT context and is loaded once by MVM-2.

The total size of the AWT context, loaded for each isolate that uses AWT, is 4920KB, of which 1312KB is attributed to the JDK's native libraries, 32KB to libawt_context.so (it includes libjnu+system.so), and the rest to X11, Xm, Xt, etc. The read-only portion of the AWT context is 3856KB. It is backed by virtual memory pages shared among contexts. Thus, a new isolate that needs to use Swing/AWT will increase the physical memory footprint by up to 1064KB due to the new AWT context's writeable memory. A process executing HSVM would consume the same amount of writeable memory for AWT.

Due to separate virtual addresses for each library, an AWT context requires almost 5MB of virtual memory. For single-user desktop systems, where a virtual machine would not typically execute many applications, this does not appear to be problematic with 32-bit JVMs. However, using this technique to run a large number of GUI programs requires 64-bit implementations of the JVM.

Memory footprint reductions due to meta-data sharing can be large, as Figure 3 indicates. The first bar shows the size of meta-data generated during the bootstrap of MVM-2. The next three bars show the amount of meta-data in MVM-2 after starting up one instance of Notepad (second bar), two instances of Notepad (third bar), and three instances of Notepad (fourth bar). The last three bars show meta-data size for the SwingSet2 demo. Meta-data is split up into the following components: (i) *permanent generation* (runtime representation of classes, including bytecodes and constant pools), (ii) *code cache* (size of dynamically compiled methods), (iii) *static state* (size of storage for mutable class components), and (iv) *class mirrors* (data associated with an instance of a class on behalf of an isolate).

Libraries	Read-only	Read-write
JDK (e.g., libmawt)	1208	**104**
X Window (e.g., X11)	2624	**952**
libawt_context.so	24	**8**
libawt_proxy.so	544	32

Table 3. Size (in KB) of components of AWT-related shared libraries in MVM-2. Underlined boldface indicates additional memory required for a new AWT context.

The first two components are the same in HSVM and in MVM-2. However, only in MVM-2 they are (transparently) shared among isolates. Thus, running concurrently three instances of Notepad in three instances of HSVM generate about 13.5MB worth of meta-data memory footprint (4.5MB for each instance). In MVM-2 the footprint is less than 5.1MB. In addition to 4.5MB generated in HSVM to run one instance of Notepad, this includes a slightly growing code cache (since more methods get compiled), constant amount of static state (explained below) and 38KB of class mirrors per isolate executing Notepad. For SwingSet2, which generates more than 7MB of metadata, the memory footprint reduction due to sharing in MVM-2 is even more pronounced.

In MVM-2 access to static fields is indirected through an array indexed by an isolate identifier. The size of these arrays is set to the maximum number of concurrent isolates MVM-2 can execute. This value is a command-line parameter of MVM-2, and in our experiments is set to 32. All such arrays comprise “static state” of an isolate. The static state component remains constant for subsequent executions of the programs if no new classes are loaded. This was the case in our experiments (Figure 3).

It is important to note that the amount of meta-data shared among different applications can be substantial, especially when they use a large JDK component. For example, Notepad run after SwingSet2 generates only

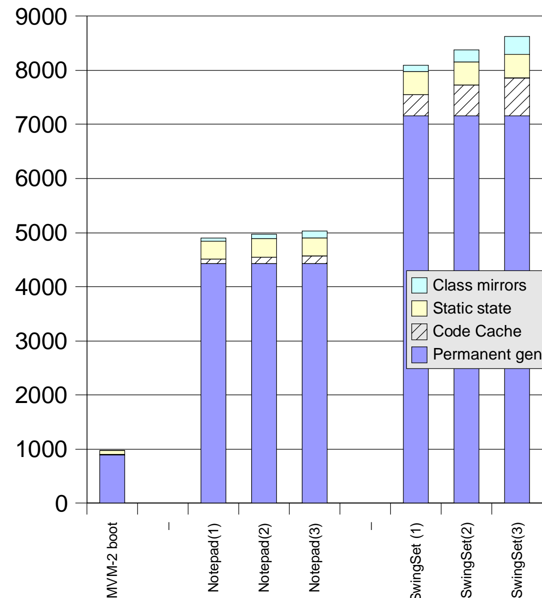

Figure 3. Memory footprint (in KB) related to meta-data of Notepad and SwingSet2.

about 100KB of additional meta data, as both applications rely heavily on Swing and SwingSet2 utilizes a large part of Swing.

Finally, the space occupied by the Jlogin processes must also be taken into account when analyzing memory footprint of MVM-2. The memory-resident image of a single instance of Jlogin that has not loaded any user-supplied native library takes 3.45 MB, 1.55MB of which are shared segments of libraries commonly used by most processes (i.e., libc, libsocket, etc.). Thus, each Jlogin process adds 1.9MB to the overall footprint. 80% of this additional space consumption is caused by the NCI library (Sec. 4), which allocates a large data structure on the process's heap. We are looking into ways of minimizing this overhead.

Interested readers are referred to [Spec98] for selected details of HSVM's performance.

5.3 Portability

The technique described in this section takes advantage of the audit libraries functionality available on the Solaris Operating Environment to map read-only segments of the library's instances to the same physical memory pages. We are not aware of any other OS with a similar functionality and a convenient interface.

On UNIX and win32 systems a library can be renamed and then loaded into a process under a different name. Thus, multiple instances of the same shared library can exist in a single process, but under different names. Their text segments will not be backed by the same physical storage. This approach can emulate our technique on modern OSes, but at a cost of enlarging memory footprint. In our settings the additional memory overhead per GUI-enabled isolate would be 3856KB (the first three entries in the "read-only" column in Table 2).

6 DISCUSSION & RELATED WORK

Our experience with the system described above is very positive: it is efficient, robust, full-featured, and fully compliant with the 1.3.1 version of the JDK.

Other projects re-architected the JVM to achieve safe multitasking, but to our best knowledge none of them offered multi-user support. KaffeOS [BHL00] is similar to MVM in its design, but much less aggressive in what data is shared. Alternative designs aimed at sharing certain meta-data among virtual machines, each of which executes in a separate process [DBC+00,CDN02]. This addresses the multi-user issue, but scales much more poorly than a single multitasking virtual machine.

In contrast to MVM-2, which simultaneously maps multiple "logical" JVMs onto a single OS process, another approach to the affinity between the virtual machine and the OS process it runs in is exploited in SAP's Process Attachable Virtual Machine (PAVM) [KKL+02]. When a user session starts, an instance of PAVM is created. It has a private session memory block, which stores the complete computational state of the session, as well as run-time data structures such as thread stacks and heap area needed by the session. This organization allows PAVM instances to mapped into any work process and then unmapped, made persistent, and eventually mapped back into the same or another work process, since the session state is completely encapsulated in the session memory. Thread scheduling is cooperative, which is beneficial for maximizing batch processing throughput. Special care has been taken to properly handle bindings to external resources by sessions.

The .NET platform [Micr02] defines *application domains*, similar to the notion of isolates. Instances of `System.AppDomain` are virtual processes isolating applications from one another. Multiple application domains can exist in a single OS process. There is no multi-user support – all collocated application domains execute on behalf of the same user. Moreover, unlike MVM-2's isolates, .NET's application domains cannot safely use arbitrary native code.

Several approaches could be applied to make native code memory-safe. However, the techniques such as augmenting native code with safety-enforcing software checks [WLA+93], statically analyzing it and proving it safe [NL96], or designing a low-level, statically typed target language to compile native code to [MCG+99] are unsuitable for our purposes. The major problem is these mechanisms add memory safety only, and do not address the issue of conflicting use of OS resources. Moreover, they either introduce non-negligible performance overheads, require source code for re-compilation, or are not general enough for complex native code. An approach applicable to MVM-2 to remove IPC costs introduced by NCI is Protected Shared Libraries [BTC97]. A protected shared library may have, within the process that loaded it, its own protection domain. Having such functionality in an OS would elegantly addresses the problem of collocating the JVM and untrusted native libraries. Our solutions, based on peer native processes to isolate untrusted code and the ability to load the same set of libraries multiple times with the same process for trusted code, are dictated by a pragmatic consideration: using the features available on a mainstream operating system.

An interesting design alternative is to re-implement an OS in the Java programming language. This would subsume certain functionality that currently requires peer native processes (e.g., instances of Jlogin) in MVM-2. For example, file access could be done entirely in the safe language. This can lead to savings and optimizations, e.g., file access permissions would be checked only once, via Java security. Hardware protection would not be required, so a part of the overhead currently due to process switching would not be incurred.

Several OSes have been implemented entirely or almost entirely in a safe language. Earlier systems, such as Cedar [SZB+86], were typically single-user. An exception is Project Oberon, which associated user identities with executing programs. However, such support was minimal, as the premise of that system was that the Oberon server "operates in a harmonious environment" ([WG92], p. 324). A more recent example is the SPIN operating system [BSP+95], implemented in Modula-3. JavaOS™ [SM99] was a first attempt to offer the complete OS functionality entirely in the Java programming language. However, the system was single-user and consisted of a single protection domain. A more complete and ambitious environment is the JX operating system [GFW+02], also written in the Java programming language.

Comparing MVM-2 with JX illustrates several points. JX does not rely at all on hardware protection. This implies that a arbitrary user-supplied native library can jeopardize the whole kernel and all applications, and is thus disallowed. In MVM-2 native code executes in a separate process, but a virtual machine bug can corrupt or abort all the current computations. The reliability of the virtual machine can improve and eventually become as robust as a kernel of a commodity OS, while a decision to entirely eliminate memory protection from the OS prevents native code from ever being run (unless the need for native code does not exist at all). Designs such as JX or JavaOS are promising but only for applications coded entirely in the Java programming language. Purity has significant advantages, though – for example, if the AWT subsystem was implemented without any native code in our base JDK, as it is done by the Remote Abstract Window Toolkit [IBM98], the issue of virtualizing core native libraries would be much less important in the development of MVM-2.

7 CONCLUSIONS

We have implemented extensions to a multitasking virtual machine that allow it to safely host isolated computations from various users. MVM-2 builds on the safety of the Java programming language to collocate multiple programs within a single-address space. The identities of users are preserved with respect to operations that need them. The virtual machine is transparently shared among computations, with improved resource utilization. Any standard API of the Java platform including file access, native code, and the graphical subsystem can be used by any computation from any user with an illusion of having the JVM all to itself.

The performance penalties for using user-supplied native code and file access operations are proportional to how often the application uses those features; in the case of remote file access only certain operations incur overhead, while reading and writing to files do not suffer any performance impact. Using the graphical subsystem does not result in any performance overhead and at the same time significantly lowers both the start-up time and the memory footprint.

In this paper we have not described any resource control mechanisms of MVM-2. This is partially due to space constraints and partially because this work is still in progress [CHS+02]. Certainly such facilities are needed before we could call MVM-2 a multi-user platform akin to an OS for isolates.

Our approach takes to the extreme the concept of the virtual machine executing on top of a commodity OS. Auxiliary processes are needed to provide multiple operating system resource and user contexts, but no modifications are required to the OS itself and at the same time no feature of Java platform is missing or compromised. We view MVM-2 as a step towards gradual blending of the functionality and implementations of virtual machines and operating systems.

Acknowledgments. The authors are grateful to Ciaran Bryce, Dave Dice, Mick Jordan, Doug Lea, Miles Sabin, Glenn Skinner, Alex Snoeren, Pete Soper, Pat Tullmann, Jan Vitek, and Mario Wolczko for their comments, suggestions and help. Special thanks are due to Rod Evans for explaining details of linkers and loaders and to Fred Oliver for sharing his insights about start-up time measurements.

Trademarks. Sun, Sun Microsystems, Inc., Java, JVM, Enterprise JavaBeans, HotSpot, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. SPARC and UltraSPARC are a trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

8 REFERENCES

[BHL00] Back, G., Hsieh, W., and Lepreau, J. *Processes in KaffeOS: Isolation, Resource*

Management, and Sharing in Java. 4th OSDI, San Diego, CA, 2000.

[BTC97] Banerji, A., Tracey, J., Cohn, D. *Protected Shared Libraries: A New Approach to Modularity and Sharing*. USENIX Annual Technical Conference, Anaheim, CA, January 1997.

[BSP+95] Bernshad, 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 SOSP, Copper Mountain, CO, December 1995.

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

[CD01] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution.* ACM OOPSLA'01, Tampa, FL.

[CDN02] Czajkowski, G., Daynes, L., and Nystrom, N. *Code Sharing among Virtual Machines.* ECOOP'02, June 2002, Malaga, Spain.

[CHS+02] Czajkowski, G., Hahn, S., Skinner, G., and Soper, P. *Resource Consumption Interfaces for Java Application Programming - A Proposal.* ECOOP'02 Workshop on Resource Management for Safe Languages, Malaga, Spain, June 2002.

[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.

[GFW+02] Golm, M., Felser, M., Wawersich, C., Kleinoder, J. *The JX Operating System.* The USENIX Annual Technical Conference, Monterey, CA, June 2002.

[GJS+00] Gosling, J., Joy, B., Steele, G. and Bracha, G The Java Language Specification. 2nd Edition. Addison-Wesley, 2000.

[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.

[IBM98] IBM Corp. Remote AWT For Java. alphaworks.ibm.com/tech/remoteawtforjava.

[JCP01] Java Community Process. JSR-121: Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp.

[KKL+02] Kuck, N., Kuck, H., Lott, E., Rohland, C, and Schmidt, O. *SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers.* Work-in-Progress Session, Java Virtual Machine Research and Technology Symposium, San Francisco, August 2002.

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

[MM01] Mauro, J., and McDougall, R. *Solaris Internals – Core Kernel Architecture.* Prentice Hall, 2001.

[Micr02] Microsoft Corp. *.NET Web Page*. http://www.microsoft.com/net. 2002.

[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*. ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, May 1999.

[NL96] Necula, G., and Lee, P. S*afe Kernel Extensions without Runtime Checking*. 2nd Symposium on Operating Systems Design and Implementation, Seattle, WA 1996.

[PCD+02] Palacz, K., Czajkowski, G., Daynes, L., and Vitek, J. *Incommunicado: Efficient Communication for Isolates.* ACM OOPSLA'02, Seattle, WA, November 2002.

[SM99] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System.* Addison-Wesley 1999.

[SZB+86] Swinehart, D., Zellweger, P., Beach, R., and Hangmann, R. *A Structural View of the Cedar Programming Environment.* ACM Transactions on Computer Languages and Systems, Vol. 8. No. 4, October 1986.

[SLN99] Schmidt, B., Lam, M., and Northcutt, D. *The Interactive Performance of SLIM: a Stateless, Thin-Client Architecture.* 17th ACM SOSP, Kiawah Island, SC, 1999.

[Spec98] Standard Performance Evaluation Corporation. http://www.spec.org.

[Stev90] Stevens, R., *UNIX Network Programming*. Prentice Hall, 1990.

[Sun00a] Sun Microsystems, Inc. *Java HotSpot™ Technology*. http://java.sun.com/products/hotspot.

[Sun00b] Sun Microsystems, Inc. *Linker and Libraries Guide*. http://docs.sun.com.

[Sun02] Sun Microsystems, Inc. *New I/O APIs.* http://java.sun.com/j2se/1.4/docs/guide/nio.

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

[WG92] Wirth, N. and Gutknecht, J. *Project Oberon.* Addison-Wesley, 1992.