Multitasking without Compromise: a Virtual Machine Evolution

Grzegorz Czajkowski Laurent Daynès
Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043, USA
+1 650 336 6501 +1 650 336 5101
grzegorz.czajkowski@sun.com laurent.daynes@sun.com

ABSTRACT

The Multitasking Virtual Machine (called from now on simply MVM) is a modification of the Java™ virtual machine. It enables safe, secure, and scalable multitasking. Safety is achieved by strict isolation of applications from one another. Resource control mechanisms augment security by preventing some denial-of-service attacks. Improved scalability results from an aggressive application of the main design principle of MVM: share as much of the runtime as possible among applications and replicate everything else. The system can be described as a 'no compromise' approach – all the known APIs and mechanisms of the Java programming language are available to applications. MVM is implemented as a series of carefully tuned modifications to the Java HotSpot™ virtual machine, including the dynamic compiler.

This paper presents the design of MVM, focusing on several novel and general techniques: an in-runtime design of lightweight isolation, an extension of a copying, generational garbage collector to provide best-effort management of a portion of the heap space, and a transparent and automated mechanism for safe execution of user-level native code. MVM demonstrates that multitasking in a safe language can be accomplished with a high degree of protection, without constraining the language, and with competitive performance characteristics.

Keywords

Java virtual machine, application isolation, resource control, native code execution.

1 INTRODUCTION

It is not uncommon to come across a computer installation where a majority of executing computations are written in the Java programming language [1,11]. The Java virtual machine [19] is used to execute a widely diversified mix of programs – from applets in Web browsers to standalone applications to Enterprise JavaBeans™ components executing in application servers. No matter whether the program is a tiny applet or a complex application, it effectively views the JVM as an ersatz operating system (OS). However, the capabilities of the environment fall short of what an OS should provide to applications. The existing application isolation mechanisms, such as class loaders [16], do not guarantee that two arbitrary applications executing in the same instance of the JVM will not interfere with one another. Such interference can occur in many places. For instance, mutable parts of classes can leak object references and can allow one application to prevent the others from invoking certain methods. The internalized strings introduce shared, easy to capture monitors. Sharing event and finalization queues and their associated handling threads can block or hinder the execution of some application. Monopolizing of computational resources, such as heap memory, by one application can starve the others.

Some of these cases of interference are subtle while others are easy to detect. Some manifest themselves rarely while others are notorious. All are undesired. Their existence perpetuates the current situation, where the only safe way to execute multiple applications, written in the Java programming language, on the same computer is to use a separate JVM for each of them, and execute each JVM in a separate OS process. This introduces various inefficiencies in resource utilization, which downgrades performance, scalability, and application startup time. The benefits the language can offer are thus reduced mainly to portability and improved programmer productivity. Granted, these are important features, but the full potential of language-provided safety and protection is not realized. Instead, there exists a curious distinction between 'language safety' and 'real safety', where the first one slips more and more into the academic domain and the other means hardware-assisted, OS-style multitasking, preferable for everyday use. While the contributions of related research are invaluable [2,3,4,5,12,24], from a pragmatic standpoint the resulting prototypes are often impractical: either the semantics of the language is constrained, some features or mechanisms are excluded, performance is unsatisfactory, or all of the above. To a large extent the proposed techniques and solutions have not yet translated into an industry-strength, widely used implementation of a multitasking-safe JVM because of a lack of a complete, well-performing prototype.

This paper presents MVM, a multitasking version of the JVM. Applications that run in the MVM are protected from one another, and sources of inter-application interaction currently present in the JVM are removed. A very lightweight design promotes sharing of as much of the runtime as possible among applications, improving performance and scalability. Unused memory is managed in a best-effort fashion, further improving performance.

The contributions of this work are: (i) the design of lightweight isolation, (ii) an approach to best-effort memory management in a multitasking environment with a shared, garbage-collected heap, and (iii) a safe method for executing user-supplied native code in such an environment. MVM demonstrates that it is possible to build a high quality, full-featured, safe-language multitasking environment.

The rest of this paper is organized as follows. Section 2 presents an overview of the goals and design of MVM. Sections 3 through 5 focus on selected details of the system: lightweight isolation, memory management, and native code execution, respectively. Selected areas of related work are discussed whenever appropriate in Sections 2 to 5.

Several recent papers [2,5,6,9] contain an up-to-date overview of related projects. We will only describe details directly related to the techniques presented in this paper, and refer interested readers to the cited work for broader exposition.

Similarly to dispersing the discussion of related work throughout the paper, the performance issues are analyzed in Sections 3 to 5. The experimental setup consists of a Sun Enterprise™ 3500 server with four UltraSPARC™ II processors, with 4GB of main memory, running the Solaris™ Operating Environment, version 2.8. The Java HotSpot virtual machine [26] (referred to from now on as HSVM), version 1.3.1, with the JDK™ version 1.3.1, is the code basis for MVM. The SPECjvm98 benchmarks [25] are used.

2 OVERVIEW OF THE DESIGN

MVM is a general-purpose environment for executing Java applications, also referred to as *tasks*[1]. MVM-aware applications, such as application server engines, can use the provided API to create tasks, to terminate, suspend and resume them at any point of their execution, and to control the computational resources available to tasks. The main task does not have to be a server – it can be any Java application.

Three goals dictate our design choices: (i) no form of interference among executing applications should be allowed, (ii) an illusion of having the JVM (with all core APIs and standard mechanisms) to itself should be provided for each task, and (iii) MVM should perform and scale well. The motivation is to make the system attractive from the practical point of view.

The key design principle of MVM is: examine each component of the JVM and determine whether sharing it among tasks can lead to any interference among them. In some cases this approach yields a clear verdict that the given component can be shared without jeopardizing the safety of the tasks. Other components are either replicated on a per-task basis or made *task re-entrant*, that is, usable by many tasks without causing any inter-task interference. This builds on the ideas described in [6]. The technique presented in that work – replicating static fields and class monitors – has been generalized in MVM to classify all components of the JVM as 'shareable' or 'non-shareable'.

Shareable components, which require some modifications to become task re-entrant, include the constant pool, the interpreter, the dynamic compiler, and the code it produces. An arbitrary number of tasks in MVM can share the code (bytecode and compiled) of both core and application classes. Non-shareable components include static fields, class initialization state, and instances of java.lang.Class. Runtime modifications presented in Section 3 make these replications transparent and ensure that the garbage collector is aware of them.

The heaps of tasks are logically disjoint, but at the physical data layout level they may be interleaved. An efficient implementation of flexible per-task heap memory guarantees, enforceable limits, and transparent re-use of surplus memory is provided (Section 4).

The separation of tasks' data sets in MVM implies that tasks cannot directly share objects, and the only way for tasks to communicate is to use standard, copying communication mechanisms, such as sockets or RMI. This is a conscious and long deliberated decision. Existing models of controlled direct sharing, such as [2,4,5,12], are not convincing from the practical standpoint. First, some of them complicate the programming model, since there are now two kinds of objects – non-shared, with unconstrained access, and shared, with some restrictions on their use. Second, some approaches have been implemented only via bytecode editing, and it is not clear what inter-task dependencies would arise from the proposed sharing if it were implemented in the runtime of a complex, high-performance virtual machine. Such dependencies can greatly complicate resource reclamation, accounting, and task termination. Finally, some proposed inter-task communication mechanisms introduce up-front overheads, affecting all tasks, even those that never communicate.

Native methods of core classes are well tested and are shared by all tasks. However, an audit of their sources was necessary to identify global state, and some global variables are replicated in MVM. For instance, in the JDK 1.3.1 a global buffer for error messages generated by native methods of the java.util.zip package is declared as:

```
static char errbuf[256];
```

In MVM the buffer is replicated so that each task using the package has its own copy to avoid accidental garbling of error messages. But two other native global variables, associated with the same package – a list of currently open zip files and an associated access lock, are not replicated in MVM, since they are a part of a global, task-safe service managing zip files. Overall, global variables in the core native libraries are rather infrequent, but the role of each of them has to be analyzed before deciding whether to replicate it or not. User-defined native code is neither known a priori nor known to be well behaved. Such libraries are not shared. A new technique to transparently execute them in a separate process is presented in Section 5.

Global state of Java classes is transparently replicated by runtime modifications (Section 3). However, an audit of core classes is still necessary to prevent interference related to using OS services. In particular, the System and Runtime classes of the java.lang package define services, which should be customized on a per-task basis. A case-by-case analysis is necessary. The resulting modifications are similar to those reported in [3,12], and we will not describe them here.

Other issues not further discussed in this paper include replicating event queues to deal with multiple tasks using the graphics subsystem, equipping each task with its own finalizer thread and finalization queue in order to avoid 'hijack the finalizer thread'

[1] The term *task* is perhaps not the most fortunate one, since it is already used in other contexts. The JSR 121 [15], under discussion at the time of this writing, is likely to come up with better terminology.

attacks, and accounting for most resources, since all of them are rather straightforward when performed in the modified runtime. The modifications to the bootstrap sequence are also not discussed, since even though many things happen between launching of the virtual machine and starting to execute an application, to a large extent they are implementation-specific and their description would not convey a general insight. Asynchronous task termination, available in MVM, is also very implementation-dependent, but considerably easier than a single thread termination, since no application state is shared among tasks. Finally, the issues related to security are beyond the scope of this paper. Let us only say that each task has its own set of permissions, and that MVM neither prevents nor clashes with user-defined class loaders and the notion of protection domains [10].

2.1 High-Level Design Choices

Such JVM evolution towards multitasking leads to a system in which the known sources of inter-task interference are removed. The examination and replication of problematic components preserves their behavior from the applications' perspective, and makes their execution in MVM indistinguishable from their execution in the standard JVM.

It is illustrative to stress the differences between this work and other approaches addressing various issues related to multitasking in the JVM. The results of related projects greatly advanced the understanding of the potential and limitations of multitasking in language-based systems. From the practical standpoint, though, existing research and resulting prototypes exhibit at least one of the following problems: (i) the black box approach, (ii) using inferior quality systems, (iii) dropping language features, or (iv) replicating OS architecture.

The black box approach. Various issues have been addressed via bytecode editing – for example, resource control [8], isolation [6,12], and termination [23]. Transforming programs to achieve desired properties is elegant, portable, and is a great tool for testing new ideas. However, treating the JVM as a black box has not yet led to a satisfactory solution to these issues. For instance, above-the-runtime isolation and resource control are far from perfect because of various interactions and resource consumptions taking place in the JVM, and task termination is problematic when a thread of a terminated task holds a monitor. Moreover, source-level transformations complicate application debugging.

Using inferior quality systems. The benefits offered by new functionality of various extensions to the JVM can sometimes exceed the associated performance and space costs. But this is often not the case, and it is important to understand the sources of overheads and the impact on application performance. Experimenting with an under-performing implementation of the JVM leads to qualitative understanding of proposed improvements, but quantitative studies do not necessarily translate into high-end implementations. Low-quality compilers may make the performance impact of a particular modification look small or negligible, and in extreme cases (no compiler, just the interpreter) the costs of modifications may be completely dwarfed by the slowness of the runtime. This distorts the cost-benefit calculus when a given feature is seriously considered for inclusion in the JVM.

Dropping language features. None of the known to us multitasking systems based on the JVM safely handles all the mechanisms of the Java programming language. This is because certain features, such as non-terminating finalizers or user-supplied native code, are particularly troublesome to make task-safe. Many applications will run correctly without these mechanisms, which is often used as a mandate to ignore or prohibit some legitimate parts of the language. From the practical standpoint this cannot be accepted. Otherwise, not all existing code can be executed in such environments, and new code has to be constrained to use only the 'approved' subset of the language. A JVM-based multitasking system must be feature-complete to avoid this.

Replicating OS architecture. Our goal is to turn the JVM into an execution environment akin to an OS. In particular, the abstraction of a process, offered by modern OSes, is the role model in terms of features: isolation from other computations, resource accountability and control, and ease of termination and resource reclamation. However, this does not mean that the *structure* of modern OSes should be a blueprint for the design of multitasking in the JVM. The runtimes of safe languages manage resources differently than an OS: memory management is automatic and object-centric; memory accesses are automatically safe; interpreted bytecodes co-exists with compiled code, and the runtime has control over which form to choose/generate. This can be taken advantage of, without undue replication of any of the mechanisms and components of the JVM.

The above are serious concerns for systems aiming at being practical, full-featured, and scalable, and that is why the design of MVM avoids all of them.

3 LIGHTWEIGHT ISOLATION

This section presents the details of lightweight task isolation in MVM. The runtime is modified according to the principle of sharing as much of it as possible among tasks and replicating everything else. The sources of complexity are: (i) class initialization, which must be done once for every task, upon first use of the class by a task [11], (ii) ensuring that the appropriate copy of a replicated item is accessed, (iii) efficient retrieval of per-task information (e.g., static variables and initialization state of classes), and (iv) making the retrieval scale with the number of tasks.

The changes affect the runtime class meta-data layout, the runtime sequence for loading, linking and initializing classes, the interpretation of a few bytecodes, the compiler, and the garbage collector. The design of MVM forces all tasks to use the same source for their bootstrap and application class loaders. This simplifies sharing of the runtime representation of classes (including bytecodes and code produced by the dynamic compiler), which is currently supported only for classes loaded by a task's bootstrap or application class loaders. This design choice does not prevent class loaders from being used in MVM, but it means that the runtime representation of classes loaded with application-defined loaders are not shared, and each such class is separately compiled.

3.1 Runtime Data Structures

The HSVM maintains two representations of a class: a heap-allocated instance of `java.lang.Class`, and an internal, main memory representation of the corresponding class file, optimized for efficient use by both the interpreter and the code generated by the dynamic compiler. The two representations reference each

other, and static variables are embedded in the internal representation of the class. Synchronization on a class object (either explicit via certain uses of synchronized blocks or implicit via synchronized static methods) is performed on the monitor of the java.lang.Class instance representing this class.

In MVM, all internal class representation information that needs to be replicated is stored in a *task class mirror (TCM)* object. Each TCM contains the static variables, a back pointer to the java.lang.Class object representing the class for this TCM's task, caches for speeding up type check operations, and the *initialization state* of the class (data indicating whether the class has been loaded, linked, initialized, etc.). Note that interference between tasks on class monitors is avoided since each task is provided a separate instance of java.lang.Class. A table of TCM objects is associated with each internal class representation. A unique task identifier indexes the table, so that each table contains one entry per task. Each entry in a TCM table consists of two references to the same TCM: the first one is set at class load-time, the second once the class is fully initialized. The reason for this will be clarified later. The internal representation of array classes is similarly changed.

Threads are augmented with a field indicating the identifier of the task on behalf of which they execute. The identifier is also used as an offset into TCM tables for fast retrieval.

Additionally, MVM maintains a table of *virtual machine* objects, each containing a copy of the global variables previously maintained by HSVM, and which must be replicated for each task in MVM. Examples of such global variables include references to special objects (e.g., instances of common runtime errors and exceptions such as java.lang.OutOfMemoryError), counters of daemons and live threads, etc. Figure 1 illustrates the organization of MVM. All TCM tables, as well as the table of virtual machine objects, initially have the same size and reflect the current maximum number of concurrent tasks that MVM can support. This maximum can be adjusted at runtime to respond to peak loads.

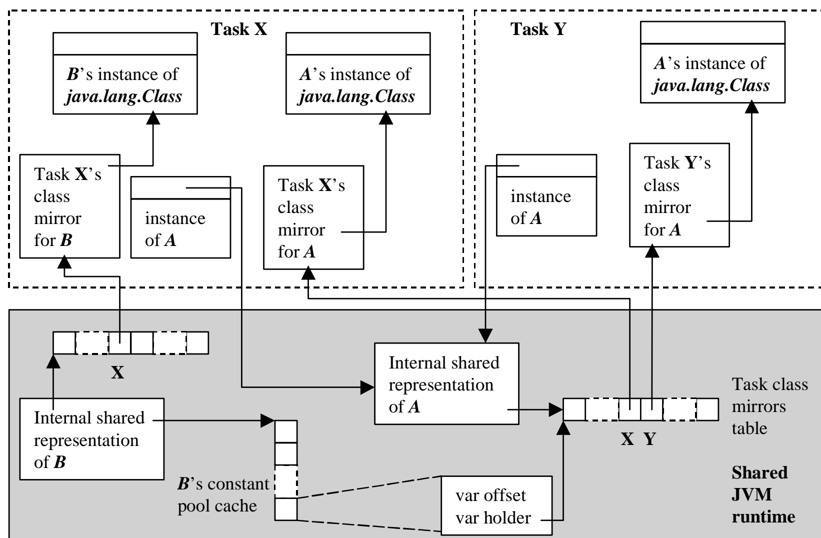

Figure 1. Organization of MVM.

For each class HSVM maintains a constant pool cache, which is a compact representation of the original constant pool, optimized for the interpreter and the dynamic compiler. Constant pool caches are built at class link time. MVM modifies the entries corresponding to static methods and static variables as follows: direct references to the internal representation of the class are replaced with direct references to their TCM table; offsets to static variables are changed to refer to offsets relative to TCMs. The interpreter uses this organization of runtime data structures to access static variables and to invoke static methods (Section 3.4).

Note that the only outgoing pointers from the shared runtime data structures to heap data of a particular task come from the objects in TCM tables, the table of virtual machine objects, and from the system dictionary (a mapping of class names to shared class representations).

3.2 Fast Class Loading and Linking

Before being used, a class must first be loaded, linked and initialized. Since in MVM most of the class representation is shared, so is the class loading and linking effort. Not all loading and linking steps can be entirely eliminated though: the JVM needs, at least, to keep track of the initialization state of a class for each task, and to carry the class loader information of a class from the time the class has been loaded. Static variables must also be accessible from other classes before the class is initialized, in order to support some legal circularity in the sequence of class initialization (it is possible for a class to see an non-initialized, but prepared, static variable of another class during the execution of a static initializer [19]).

Loading of a class consists essentially of fetching a class file from a specified source and parsing its content in order to build a main-memory representation adequate for the JVM. This needs to be done only once, by the first task to load the class, regardless of how many tasks use the class. This one-time loading sequence creates the internal representation of a class, which, in MVM, is shared by all classes, and comprises a TCM table. All the entries of the TCM table are initially set to a null value.

The task-specific part of class loading creates a TCM, prepares its static variables, enters its reference at the proper entry of the TCM table, and marks the TCM as "loaded". Class loading for all but the first task to load a class is reduced to setting up a TCM as just described.

Substantial parts of class linking can be eliminated too. Linking consists mainly of class verification, checking and updating of loader constraints, and, in the case of HSVM, building a constant pool cache for the class and rewriting its bytecodes so that they refer to the entries of the cache instead of entries of the original constant pool. In MVM all these steps need to be done only by the first task to link this class. For all other tasks, linking requires only verifying that the class and all its dependents have already been linked, and then marking the TCM "linked".

3.3 Class Initialization and Resolution Barriers

JVM implementations commonly use dynamic code rewriting techniques, such as bytecode quickening [18] and native code patching, to dynamically remove *class initialization* and *symbolic link resolution barriers*. A class initialization barrier tests whether a class has been initialized and triggers its initialization if it has not. Similarly, a symbolic link resolution barrier test whether a symbolic link (i.e., an entry in a class constant pool) has been resolved, and if not, proceeds to resolve it. Both types of barriers always succeed except the first time the barrier is encountered, hence the use of code rewriting techniques to remove them.

These barriers cannot always be eliminated in MVM because both bytecodes and the code generated by the dynamic compiler are shared by multiple tasks, which may each be at different stages of initialization for a given class. More precisely, a class initialization barrier must be executed at least once per task per class. However, most link resolution barriers need only be executed once in MVM: in order to resolve a symbolic link, the class this symbol refers to (directly or indirectly) must be loaded and linked; but, for a given class, only one link resolution can trigger its loading and linking. MVM needs a link resolution barrier for a symbol already resolved by another task only if the barrier may trigger a class load. Most bytecode instructions that require a link resolution barrier cannot trigger a class load or link by themselves, either because they are always preceded by another instruction that must have already done it (e.g., both getfield and putfield must have been preceded by a new of the corresponding object), or because they refer to constants of the current class (e.g., ldc) which must have been loaded and linked prior to the execution of the method that contains this instruction. Thus, most link resolution barriers need to be executed only once by MVM, and most of HSVM's bytecode and native code rewriting can remain unchanged.

When modifying HSVM runtime data structures, special attention was paid to minimize the overhead of both class initialization barriers and access to static variables.

Testing whether a task has initialized a class amounts to checking if the entry in the TCM table of that class for that task is non-null. On a SPARC V9 platform [30], the testing part of the class initialization barrier takes three instructions: two loads and one branch on register value.

```
ld   [gthread + task_id_offset], task_id
ld   [tcm_table + task_id], tcm
brnz,pt,a,tcm end_barrier
<delay slot filled with something useful>
call  task_class_initialization_stub
nop
end_barrier:
```

The first load fetches a unique internal task identifier from the current thread data structure (a register is allocated to permanently store the address of this structure). The task identifier is then used as an index into the TCM table, to fetch the address of the corresponding TCM (second load). A null address means that the class has not yet been initialized by the current task, and a branch on null register value appropriately dispatches to the interpreter runtime for initializing the class for the current task.

Augmenting TCMs with fields recording the initialization status of the class and the thread initializing this class for the corresponding task is enough to make the existing HSVM code task-re-entrant. The main issue is to locate the TCM of the initializing task. The entry in the TCM table cannot be used to store the TCM created by the task during class load but before the class is initialized for this task, because this would invalidate the null pointer test performed upon a class initialization barrier. This is why the TCM table holds two references to the same TCM for each task. As illustrated by Figure 2, each reference is set up at a different time: the first one is set up during class loading, whereas the second one is set up once the class is fully initialized. Class initialization barriers test the first entry only: when the test performed by the barrier fails, the TCM for the current task can be obtained simply.

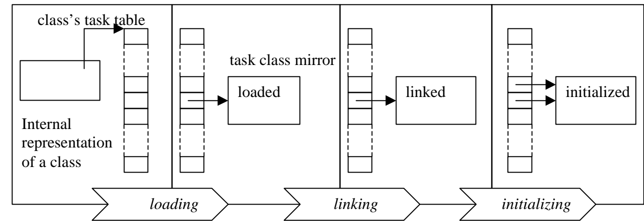

Figure 2. Fast initialization testing.

3.4 Bytecode Interpretation

MVM leaves the interpretation of all standard bytecodes unchanged. Changes are required only for the interpretation of some of the quick versions of standard bytecodes, and for the handling of synchronized static methods, which requires finding the instance of java.lang.Class that represents the class for the current task in order to enter its monitor.

Quickened bytecodes result from the execution of standard bytecodes that require a link resolution or a class initialization barrier. The interpretation of such standard bytecodes happens only once and typically consists of executing a barrier, rewriting the bytecode instruction with a quick version of it, and executing the quick version. In particular, the standard getstatic and putstatic bytecodes don't access static variables; this is done by their quick versions.

As explained in section 3.3, all class initialization barriers that are eliminated by bytecode quickening need to be re-introduced. This affects four bytecodes only: the quick versions of new, invokestatic, getstatic, and putstatic. The first two require an extra load instruction before the barrier code described in section 3.3, in order to fetch the task table of the class. This increases the path-length of these bytecodes with 4 instructions. The quickened versions of getstatic and putstatic need, in addition to the class initialization barrier, access to the TCM of the current task to access the static variables of the class. A cost-free side effect of the barrier is to set a register to the TCM of a class. Thus, once the barrier is passed, the static variable can be obtained with a single memory load. This adds 3 instructions to the path-length of getstatic and putstatic.

3.5 Sharing Compiled Code

Although our practical experience is based on the *client* compiler of HSVM (known as Compiler 1, or C1), the principles described below are generally applicable to most runtime compilers. HSVM invokes C1 to compile a method to native code once the method

has reached some usage threshold (expressed as a number of method invocations). The threshold itself is chosen to correspond to a trade-off between the cost of compilation and the perceived performance pay-off. C1 operates in two steps. First, it generates, into a code buffer, a version of the code that assumes that all classes used by the method have been resolved and initialized. This means that the code thus produced does not contain any link resolution or class initialization barriers. The second step scans the code buffer and evaluates the assumptions made. If the assumption is invalid (e.g., the class has not been initialized), the optimized code is copied into a special area at the end of the generated code and then replaced with a call to a patching stub that will properly call the JVM runtime. Patching stubs arrange for proper saving of registers and enforce runtime calling conventions, such as setting up garbage-collection safe-point information. Calls to the JVM runtime from a patching stub resume by atomically writing back the optimized code.

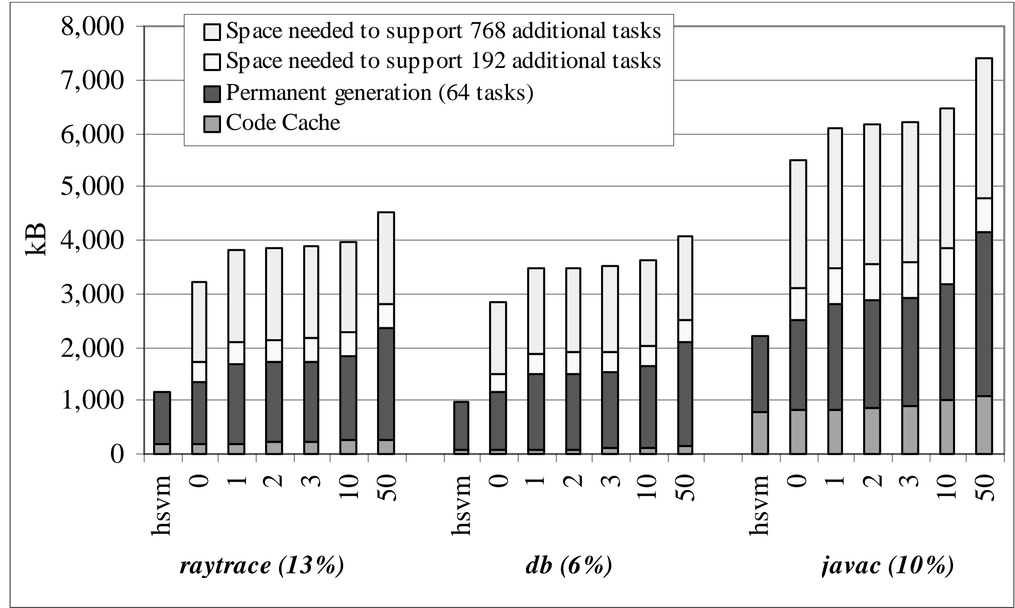

Figure 3. Memory footprint of MVM runtime. Numbers in brackets indicate the percentage of the total footprint attributed to report the percentage the code cache and permanent generation

This code patching mechanism is problematic in MVM because it dynamically eliminates class resolution and initialization barriers. As discussed earlier in section 3.3, the elimination of some of these barriers by one task may be incorrect for another one. We opted for a simple solution whereby class initialization barriers are always planted in the generated code. Only those link resolution barriers that could trigger class loading and linking are inserted (Section 3.3). These barriers are planted even for classes that are already loaded and initialized at the time the method is being compiled. Behind its apparent simplicity, this approach has a major advantage over more complex schemes that allow elimination of class initialization barriers: because the generated code is re-entrant, new tasks entering the system can immediately start executing the native code of a method already compiled by other tasks, without having to interpret the method at all. The downside is that the compiled code is sub-optimal because of the barriers left in it.

Under this approach, making HSVM task re-entrant requires only three types of changes to the code generated by the compiler. First, manipulation of static variables is changed to include the level of indirection introduced by multitasking. Second, code generated to enter/exit the monitor of a class is modified to locate the appropriate instance of java.lang.Class. Third, for all bytecode instructions that require a class initialization or link resolution barrier systematically, the compiler has been modified to generate code that handles two different cases: (i) at compile time, the class has never been loaded by any task, and (ii) the class has already been loaded by at least one task. The sequence of instructions generated for the first case is essentially a call to a patching stub that will be overwritten at runtime with another sequence of instructions handling the second case, using the patching mechanisms already used by HSVM. The difference is that this time, the code generated by the patching stub includes a barrier. The code for a barrier, and for accessing static variables, is the same for the interpreter and the compiled code, and has been described in Section 3.3.

3.6 Performance

Two performance goals have steered the design of lightweight isolation: (i) reducing the per-program footprint of the JVM runtime in order to better scale with the number of tasks; and (ii), amortizing across program executions the overhead incurred by the runtime itself, principally virtual machine startup, dynamic loading, and runtime compilations.

The bulk of the runtime data of HSVM resides in two memory areas: the code cache, where native code produced by the dynamic compiler is stored, and a portion of the heap known as the permanent generation, which mainly holds the runtime representation of class files (class metadata, constant pool, bytecode, etc.) and other runtime data structures. MVM's design of lightweight isolation seeks to share between tasks the parts of both areas that correspond to core and application classes (i.e., classes whose defining loader is not an application-defined class loader). Figure 3 compares the memory footprint of MVM with that of HSVM for three benchmarks from SPECjvm98 (raytrace, db, javac). Each bar reports the accumulated sum of the code cache size and the permanent generation size for execution of a benchmark with HSVM (bars labeled "hsvm"), or for execution of multiple instances of the same benchmark by concurrent tasks with MVM (bars labeled with a numeric value indicating the

number of tasks; a value of 0 means that the program was run as the main task; a value n>0 means that an application manager task controls the execution of n programs, each executed as a task). MVM is started with a parameter indicating the initial number of tasks it can support. This parameter determines the default size of the TCM tables that are part of the runtime representation of classes in MVM. Three configurations were used for our measurements: 64, 256, and 1024. The permanent generation portion of each bar includes the whole of the permanent generation sized for 64 tasks. The two topmost components of bars related to MVM indicate how much additional space is needed to support, 256 and 1024 tasks (i.e., 64+192, and 64+192+768 tasks, respectively). TCM tables only consume this additional space. Since none of the benchmarks uses application-defined class loaders, the numbers reported for MVM reflect exactly the size of the shared part of both the permanent generation and code cache.

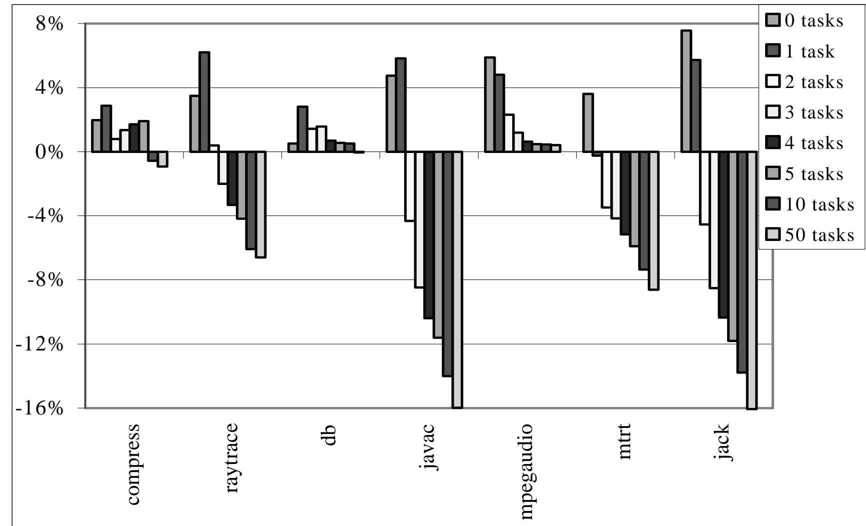

Figure 4. The performance of MVM. Time overheads are reported as relative to HSVM.

Several observations based on Figure 3 can be made. First, the code cache can contribute to up to 36% (javac) of HSVM runtime footprint, and is therefore worthwhile to share. Second, the size of MVM's shared code cache increases with the number of program executions, as the number of compiled methods increases, although this increase is very modest. Similarly, the size of the permanent generation also slightly increases with the number of running tasks. This increase corresponds to the TCM objects (mostly, storage for static variables) added by each running task. Although these are allocated in the old generation, we reported them here for comparison with HSVM, which holds the corresponding information in its permanent generation.

In contrast, running n programs with HSVM would require n times the amount reported on Figure 3 for HSVM. The space saving with MVM is immediate: as soon as 2 programs are executed when MVM is sized for 256 tasks or less, and as soon as 4 programs are executed when MVM is sized for 1024 tasks. Savings in space quickly reaches one order of magnitude as the number of programs increases.

These numbers have to be contrasted with the overall footprint of a program (total heap + code cache): across the benchmarks used for Figure 3, the runtime data of HSVM (code cache + permanent generation) amounts to between 6% (db) to 13% (raytrace) of the program maximum memory footprint.

Sharing the JVM runtime can also improve performance in many ways. First, application startup time is improved when compared to HSVM, because several steps necessary to launch an application are substantially shortened (e.g., loading and linking of all the bootstrap classes), or completely eliminated (e.g., the initialization of many internal runtime structures, such as heap, dictionary of loaded classes, etc). Only the first application, typically an application manager, will pay all these costs. The application execution time, defined here as time needed to execute the static `main` method (the application entry point), also benefits from similar savings (i.e., loading and linking of classes already loaded and linked by other tasks is faster). More importantly, sharing compiled code means that a task can immediately re-use the code of methods compiled by other tasks, thus shaving off compilation and interpretation costs from its execution time. Furthermore, in MVM, methods infrequently executed by a single execution of a program may still be compiled after several executions of the same or other programs. This is so because method invocation counters are global in MVM. Thus, the code quality may be continually improving, amortizing earlier efforts. This benefits code of popular applications as well as frequently used core classes.

On the other hand, multitasking introduces class initialization barriers, and a level of indirection when accessing static variables. The code produced by the dynamic compiler is typically bigger in MVM than in HSVM, because of barriers and the associated stubs to call the runtime. Since the costs of some of the compiler operations depend on the number of these stubs and the size of the code produced, MVM indirectly increases the costs of dynamic compilation.

Figure 4 reports results from experiments that illustrate the trade-off between the costs of making the JVM task re-entrant and the performance benefits of multitasking. The experiments consisted of executing a program repeatedly as a sequence of tasks. For each sequence, measurements are reported as time overhead (or improvement) relative to HSVM. Bars labeled with 0 report the overhead of running one instance of a benchmark in MVM as a main application. Bars labeled with a value n>0 report the overheads of running n benchmarks as n tasks started by a simple application manager (the manager is the main application in this case). When n>0, the data reported is the overall execution times of MVM divided by n. In other words, the overhead reported includes the overhead of the application manager task. When only one task is executed, the overhead of the application manager task is not amortized, and performance is usually worse than when no application manager is used (bars corresponding to "0 tasks").

The overheads of this initial prototype of MVM are within 1-7% when only one instance of a given benchmark is run. As soon as more instances of the same code are executed, though, MVM outperforms HSVM in several cases, and remains under 2% of overhead in all cases. The performance gain grows with the number of executed applications, mainly because of the

elimination of runtime compilation and interpretation due to the immediate availability of compiled code. After 50 tasks, the *average* execution time can decrease by as much as 16%.

4 MEMORY MANAGEMENT

An ability to manage computational resources is necessary to guarantee certain amounts of resources needed for tasks, and to foil certain denial-of-service attacks. Controlling and managing heap memory is one of the hardest problems, if not the hardest one, in this area, mainly because of the difficulties associated with revoking or reclaiming the resource from an uncooperative task.

In MVM, all memory of a task is reclaimable once the task has completed. Each task has a guaranteed amount of memory. Tasks are not allowed to allocate more memory than their guaranteed limit. The only exception to this is temporary, transparent use of surplus memory (i.e. memory not backing up any guaranteed amount – Section 4.1). Reclamation of surplus memory from a task is transparent, non-disruptive, and efficient. Accounting for memory consumption, needed for enforcing the limits, is accurate and introduces neither performance nor space overheads (Section 4.4).

The associated API allows privileged tasks, such as an application manager, to set limits on how much memory a task can use. The limits can be dynamically changed, in order to improve overall memory utilization. A privileged task can also set overuse callbacks [8], which are triggered whenever a particular task attempts to violate its memory limit. The callbacks decide what to do next with an offending task; one option is instantaneous termination. Privileged tasks can control the amount and the recipients of surplus memory, by assigning surplus memory priorities to tasks.

Only minor modifications to HSVM's heap management were necessary to incorporate the mechanisms needed for memory accountability, flexible management of surplus memory, and per-task garbage collections. In HSVM the application portion of the heap is organized as two generations, old and new. The new generation follows a design suggested in [28], and is sub-divided into the creation space (*eden*) and aging space, which consists of a pair of semi-spaces – *to* and *from*. New objects are allocated in eden, except for large arrays, which are allocated directly in the old generation. When the new generation is garbage collected (scavenged), the survivors from eden and the objects from the from-space whose age is below a certain threshold are copied into the to-space. Mature objects (i.e. with age above or equal to the threshold) are copied to the old generation. The age of each object remaining in to-space is incremented. Finally, the roles of to-space and from-space are swapped. Whenever the old generation fills up, a global four-phase pointer-forwarding mark-and-compact collection is triggered.

MVM gives to each task its own private new generation, consisting of the to-, from-, and eden spaces. The old generation remains shared among all tasks. The rationale for these choices is that most action takes place in the young generation – the vast majority of objects are allocated there, and garbage collection happens there much more frequently than in the old generation. Multiplying the number of young generations is simple and eliminates most of the heap-related interference between tasks. The implications of this decision are further discussed in Section 4.3.

4.1 Management of Surplus Memory

Surplus memory is memory not used at the moment as a part of guaranteed heap space for any task. Using it may improve application performance. For short-lived tasks, adding memory can postpone GC until after the task completes and all of the task's memory can then be quickly reclaimed, potentially avoiding any GC activity. For long-lived, non-interactive programs, additional memory can reduce the frequency of GC. Using surplus memory may thus improve performance but can also lead to increased collection pause times. This is undesirable for some applications, such as the interactive ones, and may be avoided by assigning an appropriate surplus memory priority.

Modern automatic memory management systems typically copy data to make allocation faster and to improve data locality. This is often combined with generational techniques, where long-lived (also known as old or tenured) data is moved around less frequently, and new (young) objects are allocated in the new generation, where they either die quickly or are copied and finally get promoted (tenured) to the old generation. Such designs optimize for the weak generational hypothesis, which states that most objects die young [28]. The new generation is a good candidate for receiving surplus memory. Since allocations take place there, more memory available means more allocations without triggering a collection. Less frequent collections allow more objects to die before collections, which can improve performance. Surplus memory can be easily integrated into, used by, and promptly cleared of data and taken away from the new generation.

The memory for new generations is under the control of the New Space Manager (NSM). NSM ensures that each newly started task has a new generation, and is in charge of the best-effort management of currently unused memory. Whenever an allocation fails because of eden being full, instead of triggering new generation collection, NSM may give the allocating task another memory chunk, extending eden. Thus, each new generation now consists of a from-space, to-space, and a linked list of eden chunks. The last chunk is called *current eden*, and new

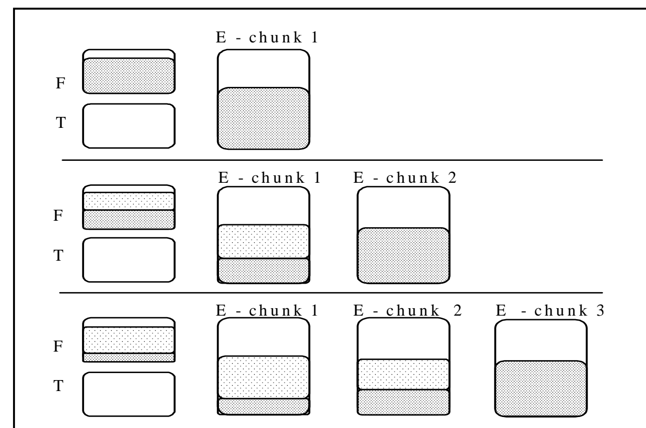

Figure 5. Extending eden in MVM. Dark shade shows live objects; light shade shows objects that died and did not have to be copied because of best-effort surplus memory management in MVM.

object allocations are performed there.

The ability to grow eden may postpone the need for a scavenge, letting more objects become garbage before a collection takes place. Figure 5 explains this: a full initial eden chunk is extended, avoiding a scavenge (top of the figure). When the second chunk is full (middle), a third one is given to the task, further postponing the collection. Finally, the third chunk fills up and NSM either does not have more chunks or the task's surplus memory priority does not allow further extensions. The scavenge is necessary (bottom). Delaying a collection until then avoids the copying of objects that have been live after the first or second extension but died before the scavenge, and avoids copying some non-tenured objects back and forth between from-space and to-space, because the scavenge did not take place twice. The age of objects is adjusted accordingly (e.g. in our example the ones in from-space will age by 3, the ones in the first chunk by 2, etc), but only during the actual copying of objects out of eden or out of from-space. Extending eden ages objects implicitly, based on what chunk they are in.

Because the cost of scavenging depends on the number of live objects and not on the eden size, this scheme can improve the performance by decreasing the total time spent in scavenging. It can also increase the times of those scavenges that actually take place. Also, since this scheme spreads the objects to be scavenged over a larger memory area, different locality properties with respect to caches hold than in the original scheme, which can degrade performance. For these reasons a task can opt out of this scheme.

4.2 Memory Accounting

The memory usage counter of a task keeps track of the amount of the old generation occupied by the objects of this task. It also includes the sizes of from-space, to-space, and the initial eden chunk of the task, regardless of how many objects are in the new generation, because these spaces are always reserved for the task. The value of the counter is incremented only during large array allocations or upon promotion of young objects to the old generation during a scavenge, since only then does the volume of the task's data in the old generation increase. It is decremented during global collections, to reflect the actual space taken by lived tenured objects of that task. The sizes of eden extensions introduced by surplus memory management are not included in the usage counter because they are temporary and meant to improve performance when there is no contention for memory.

MVM isolation properties guarantee that data sets of different tasks are disjoint. Thus, the collection roots of different tasks are disjoint too. The garbage collector is restructured to take advantage of this property. During global collections the marking starts from all the roots of the first task, then the second task's objects are marked, and so on. It is thus inexpensive to determine how much live data the task has after the collection. New generation collections do not need any additional re-ordering – whenever an object is tenured, it is known which task's new generation is collected, and the appropriate counter is incremented. No accounting takes place at object allocation time (Figure 6), since the size of the new generation is already included in the value of the usage counter.

Whenever the usage counter is incremented, it is compared against the task's heap memory limit. An old generation collection is started when this increase makes the usage counter exceed the

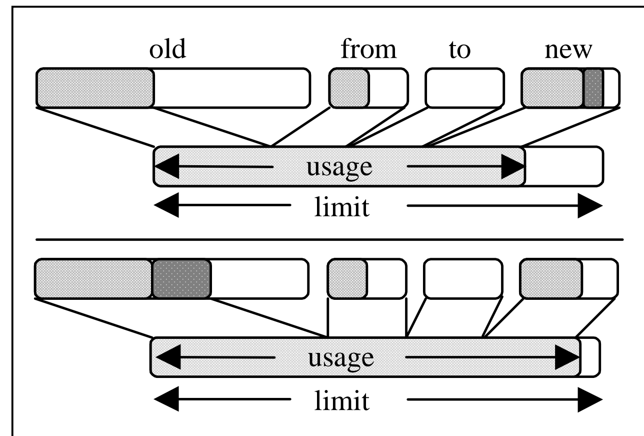

Figure 6. Task's memory usage counter includes the size of new generation (to, from, and initial eden chunk). An allocation of an object in eden does not change the usage (top), but an allocation in the old generation does (bottom).

limit. If after the collection the new allocation would still exceed the limit, an overuse callback is triggered. Similarly, when during surplus memory reclamation it is discovered that the total size of task's data is larger than the limit, the overuse callback is triggered.

4.3 Interference among Tasks

MVM is free of two memory management-related interference issues troubling the JVM: a task cannot hijack the finalizer thread (finalizer queues and finalizer threads are replicated in MVM), and it cannot monopolize heap memory (Section 4.1). Since each task has its own new generation, allocations there are independent of these performed by other tasks. The surplus memory management introduces one shared service, NSM. This also does not lead to inter-task interference, since NSM is called infrequently (to request or return an eden chunk); the calls are non-blocking (simple management of relatively large chunks); and requests for chunks return immediately when no surplus memory is available.

Interferences due to allocations in the shared old generation are not a concern, because the old generation is compacted, which enables very fast linear allocation with non-blocking synchronizations. Moreover, allocations in the old generation happen in bursts and infrequently, since they result from typically rare promotions that take place during scavenges.

Scavenges and global collections cause interference because they currently require all threads to be suspended. In principle, this is not required for scavenges, since only threads of the task being scavenged really need to be suspended. Ideally an incremental collector should be used for the old generation in MVM, in order to enable independent collections. For instance, a train algorithm [14] seems like a good match, since each car could be indivisibly allocated to a task and independently collected.

4.4 Performance

Performance overheads introduced by memory management in MVM are minimal, and come from three sources. First, the

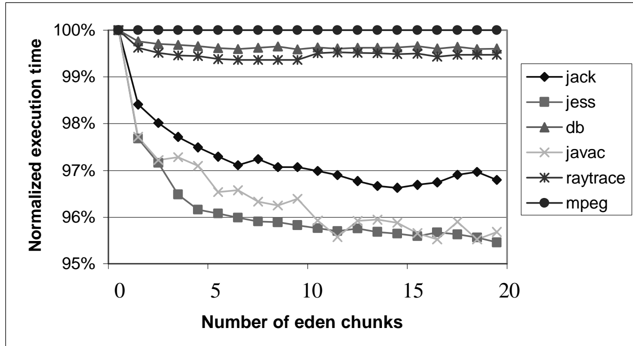

Figure 7. Execution time as a function of an eden size in MVM.

overheads of replicated new generations are virtually zero (they do not record in our measurements). This is so since only one indirection, to fetch a reference to the task's new generation, is added to object allocations. Second, the overheads introduced by memory accounting do not exceed 0.5% for any benchmark. The overheads are proportional to the number of promoted objects – each promotion includes adding the already computed (HSVM) object size to a per-task counter and then comparing the new value with a limit. Most objects get promoted in *javac* (almost two million for 2MB of eden space) and the overheads incurred in this case are 0.5%. Many fewer objects are promoted in other benchmarks – from a few thousand (*compress*, *mpeg*) to about half a million (*jack*), which explains why the overheads are negligible. Finally, the costs of extending eden and associated aging-related bookkeeping during promotions are also low. Overall, total memory-related overheads do not exceed 1% of execution time.

To quantify the benefits of surplus memory management, all benchmarks were run with an MVM eden chunk size equal to 512KB (larger sizes yield similar results; smaller are impractical). It is also the initial size of eden. The parameter of the experiment was the number of chunks the application can use. The maximum number of chunks was fixed for the whole run of the benchmark, so effectively the union of all eden chunks formed eden, managed by NSM.

The execution times are presented in Figure 7 (*mtrt* is omitted, since it is similar to *raytrace*; *compress* is omitted, since the results were very similar to *mpegaudio*). Each benchmark was run in MVM, and the baseline for comparisons was the run with one, initial chunk only, but including all the costs of memory accounting.

The unsurprising conclusion is that the execution time can benefit from larger edens. The gains can be as high as 4% (*javac*, *jess*), but, as surplus memory insensitive mpeg shows, they may just not exist.

A task can get more eden space for only a portion of its execution time, and still benefit from it, although less than if the eden was permanently extended. Overall, the scheme is simple and inexpensive, and may bring about a few per-cent performance improvements when there is surplus memory available. In view of the fact that HSVM is very well tuned, an ability to gain even 2% improvement is attractive.

5 USER-LEVEL NATIVE CODE

The coexistence of programs written in a safe language with user-supplied, unsafe (native) code is convenient (it enables direct access to hardware and OS resources and can improve application 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 can be much more serious. In addition to causing malfunctioning of the loading application, such a library may corrupt the data of other applications or crash the whole virtual machine. Worse yet, an uncontrolled malicious native library may read the data of other applications, perhaps leaking out privileged information.

Several techniques for ensuring memory safety have been proposed, such as augmenting native code with safety-enforcing software checks [29], statically analyzing it and proving it safe [21], or designing a low-level, statically typed target language to compile native code to [20]. 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. In particular, ensuring memory safety with these techniques requires source code of native libraries [20], generating safety proofs [21], which is impossible in general, or may incur non-negligible performance penalties [29].

Memory safety is not the only issue, though. Guaranteeing the safe 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) [17], which is its sole interaction with 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, and therein lies a problem. The JVM makes certain decisions regarding the use of the host OS interface and of available resources. Examples include the following: signal handlers may need to be instantiated to handle exceptions that are part of the operation of the JVM (e.g., to detect memory access and arithmetic exceptions, etc.); the JVM must choose a memory management regime for its own purposes, such as the allocation of thread stacks and red zones; threads accessible in the language are typically mapped onto the underlying system's threading mechanism; the JVM adopts a convention to suspend threads for garbage collection; the JVM decides how to manage I/O (e.g., using blocking or non-blocking calls).

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, put them together into one virtual machine, and expect the resulting system to work correctly. The implicit conventions in which the JVM uses system resources are rarely documented, are highly dependent on the implementation decisions, and are usually thought of as private to the JVM. Thus, while programmers may have very legitimate reasons, for instance, to customize signal handling in a native method, doing this may interfere with the JVM, depending on unknown implementation details.

In MVM user-supplied native code is executed in a separate process. Each task that needs this 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 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.

The challenges were to make this execution of native code in a separate process transparent to the native libraries and to tasks. We also wanted to avoid any modifications to the JVM, so that our design can be easily reused in contexts other than multitasking.

5.1 JNI Essentials

JNI interacts with the JVM via *downcalls* (when a Java application calls a native method) and *upcalls* (when a native method calls up to 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, the names of which are generated, according to a known convention, from the names of Java methods declared as native. 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 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.

5.2 The Design of Native Code Isolation

Native code isolation works by interposing an isolation layer between the JVM and native method libraries such that native methods are transparently executed in a process different from the process executing the JVM. For simplicity of exposition, let us call the former *n-process*, and the latter the *j-process*. Figure 8 gives a high-level view of the interposition mechanisms. The interposition layer maintains one n-process per task actually issuing native method calls, and directs all native calls issued from a task to its corresponding *n-process*. Let us assume a simple scenario where one task performs calls to one native library, called hereafter *l-orig*.

A native library with the same name and exported symbols as *l-orig* is produced so that it can be loaded by the Java application executing in *j-process*. Let us call this library *l-proxy*. It is generated through an automated analysis of the symbol table of *l-orig*. All extracted function names that comply with the downcall naming convention are used to generate a source file in which these functions are redefined to ship 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. Prior to that, *n-process* replaces the first word in the list of received arguments with its own version of the JNI environment pointer. This custom JNI environment redefines all JNI functions so that each of them ships all of its arguments along with its unique identifier back to *j-process*, where the upcall is dispatched to the original JNI method. Generating proxy libraries and replacing the original JNI environment pointer with a custom one are two main techniques making the resulting interposition mechanism transparent to the JVM and to native libraries.

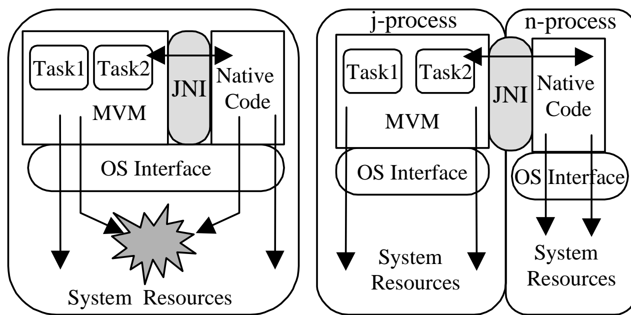

Figure 8. Various conflicts are possible between user-supplied native code and the JVM (left). In MVM, the native code isolation scheme prevents them (right).

Implementing the upcalls in the custom JNI environment interface is straightforward for those taking a fixed, known number of arguments of primitive or opaque reference or identifier types. Invoking methods or constructors is handled by JNI upcalls that expect a variable number of arguments. Each such upcall has three forms: (i) using the "…" construct of the C programming language, (ii) expecting arguments as a variable-length list (`va_list`), and (iii) expecting a vector of unions of a JNI-defined type, each of which holds one argument for the upcall. Computing the number of the upcall's arguments at call time by analyzing the current stack call frame is possible only in the first case.

An elegant, portable, and uniform solution applicable to all three cases takes advantage of the following fact. Before invoking a Java method via a JNI upcall, the method identifier has to be obtained first. This is done by calling a JNI upcall (e.g. `GetMethodID`) and supplying it with the signature of the Java method or constructor to be executed by the upcall. The number and size of arguments is obtained by analyzing the signature after successfully obtaining the method identifier. The original method identifier and the computed argument information are then cached by *n-process* for later use.

It is important to ensure that upcalls be handled in the context of the thread that originally issued the downcall. For instance, an exception thrown in an upcall has to be dispatched to the thread that caused the downcall; requests to obtain a stack trace should look identical to those generate by traditional JNI systems; monitor acquisitions by native code should be done in the context of the same thread that issued the downcall, or synchronization between native and Java code may be impossible. How this is guaranteed depends on the inter-process communication chosen. Details on this aspect of native method isolation are extensively discussed in [7].

5.2.1 Portability

The implementation of native code isolation is highly portable across different hardware/OS platforms. There are two places, which require platform-specific code: (i) determining 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*. The reason is 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. On our platform, the first issue is handled by analyzing the stack frame of a downcall. The second is dealt with by obtaining, after executing a downcall in *n-process*,

the contents of both registers (`%o0` and `%of`) that may contain the return value of the downcall. These values are then restored in *j-process,* after a return from the proxy call. Since the calling convention requires spilling of these two registers before function calls, this approach is correct.

5.3 Performance

A key factor for the performance of native method isolation is the inter-process communication mechanism chosen between the JVM process and the native method servers. Our implementation uses *doors* [13], a fast inter-process communication mechanism available on the Solaris Operating Environment. Doors achieve low latency by transferring control back and forth between the caller's and the callee's threads directly, without passing by the scheduler.

Table 1 summarizes the overheads. The "downcall" column reports the cost of a trivial downcall (no arguments, immediate return), while "downcall + upcall" additionally issues the *GetVersion* upcall, which returns immediately with an integer specifying the JNI version. 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 large indeed, for downcalls that do not compute much. To see how quickly the overheads become tolerable, let us analyze a simple program that performs $A = A^2$ for a floating-point matrix A. The squaring requires one downcall (to initiate the native call) and two upcalls (to fetch the matrix entries and to set the computed result). Figure 9 summarizes the overheads as a function of array size (e.g. size 25 means a 25x25 array). For small arrays the overheads are prohibitive. For 40x40 arrays they become tolerable, and virtually disappear when the array size approaches 90. These numbers are no substitute for more realistic benchmarks, but serve as a useful estimate of the costs of this approach. Overall, the overheads depend on the frequency of native calls issued by an application.

The proposed scheme enables safe, reliable, and interference-free composition of native libraries and the MVM runtime. No changes to the MVM were necessary as the infrastructure is transparent to and independent of the implementation of the JVM, its vendor or version. The transparency and automation of the presented technique are major improvements over such designs Microsoft's Common Object Model (COM) [22], where components can execute in the same process or in a separate process. This is accomplished with the help of an interface definition language (IDL), used to describe data passed into and

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

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

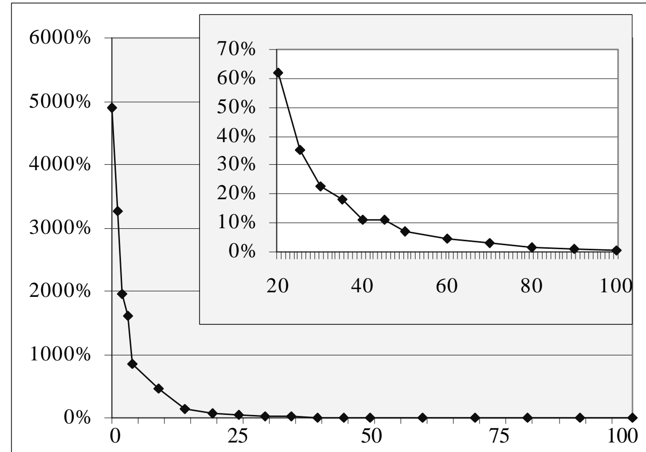

Figure 9.The overheads of squaring a matrix in a native method in a separate process, as a function of matrix dimension, relative to standard, in-proc JNI.

out of out-of-proc components.

The utility of our approach reaches beyond MVM [7]. For instance, we are using it for mixed-mode debugging, where a C/C++ debugger (e.g. gdb) executes *n-process* and the Java code is debugged with a Java debugger (e.g. Forte™ for Java [27]).

6 CONCLUSIONS

MVM is a complete system, where any existing Java application can execute and can use all abstractions, mechanisms, and standard libraries of the JVM. Known sources of inter-task interference, present in the JVM, are removed in MVM. The safety, scalability and uncompromising approach to offering each feature of the language make MVM an attractive platform for environments that require the execution of numerous programs written in the Java programming language, such as, application servers, extensible web servers, or even user desktop environments.

Special attention has been paid to three issues. First, sharing the native code that results from runtime compilation of methods of both core and application classes. This is crucial for performance, since it amortizes compilation costs across all tasks, and scalability, since it increases the shareable portion of a program memory footprint. Second, heap memory, a resource notorious for unaccountability and irrevocability in the JVM, is subject to explicit guarantees and is inexpensively accounted for. Moreover, surplus memory is transparently given to task whenever available, to improve overall performance. Third, native libraries – a feature typically forbidden in safe language multitasking environments – can be safely used by tasks, and transparently execute in a separate OS process, to prevent any interference with MVM.

The code base for implementing MVM is the Java HotSpot virtual machine. The overheads introduced by multitasking rapidly disappear or, in many cases, are replaced by significant performance gains, as repeated executions of methods enjoy accumulated compilation effort.

This paper is an initial report on MVM. The system has just become operational, and more study is needed (and planned!) to evaluate various aspects of its performance and functionality,

especially under very large workloads such as those incurred by application servers.

7 ACKNOWLEDGEMENTS

The authors are grateful to Godmar Back, Dave Dice, Robert Griesemer, Wilson Hsieh, Mick Jordan, Hideya Kawahara, Peter Kessler, Doug Lea, Tim Lindholm, Nate Nystrom, Fred Oliver, Glenn Skinner, Pete Soper, Ricky Robinson, Pat Tullman, Dave Ungar, and Mario Wolczko for their comments, suggestions and help.

8 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.

9 REFERENCES

[1] Arnold, K., and Gosling, J. The Java Programming Language. 2nd Edition. Addison-Wesley, 1998.

[2] Back, G., Hsieh, W., and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. 4th Symposium on Operating Systems Design and Implementation, San Diego, CA, 2000.

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

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

[5] Bryce, C. and Razafimahefa C. An Approach to Safe Object Sharing. ACM OOPSLA'00, Minneapolis, MN, October 2000.

[6] Czajkowski, G. Application Isolation in the Java Virtual Machine. ACM OOPSLA'00, Minneapolis, MN, October 2000.

[7] Czajkowski, G., and Daynès, L., and Wolczko, M. Automated and Portable Native Code Isolation. Sun Microsystems Laboratories Technical Report, TR-01-96, April 2001.

[8] Czajkowski, G., and von Eicken, T. JRes – A Resource Accounting Interface for Java. ACM OOPSLA'98, Vancouver, BC, October 1998.

[9] 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.

[10] Gong, Li. Inside Java 2 Platform Security. Addison Wesley, 1999.

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

[12] 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.

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

[14] Hudson, R., and Moss, E. Incremental Collection of Mature Objects. International Workshop on Memory Management, September 1992.

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

[16] Liang S., and Bracha, G. Dynamic Class Loading in the Java Virtual Machine. ACM OOPSLA'98, Vancouver, BC, Canada, October 1998.

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

[18] Linholm, T., and Yellin, F. The Java Virtual Machine Specification. 1st Ed. Addison-Wesley, 1996. Also: java.sun.com/docs/books/vmspec. Discusses bytecode quickening.

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

[20] 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.

[21] Necula, G., and Lee, P. Safe Kernel Extensions without RuntimeChecking. 2nd Symposium on Operating Systems Design and Implementation, Seattle, WA 1996.

[22] Rogerson. D. Inside COM. Microsoft Press, 1997.

[23] Rudys, A., Clements, J., and Wallach, D. Termination in Language-based Systems. Network and Distributed Systems Security Symposium, San Diego, CA, February 2001.

[24] 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.

[25] Standard Performance Evaluation Corporation. SPEC Java Virtual Machine Benchmark Suite. August 1998. http://www.spec.org/osg/jvm98.

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

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

[28] Ungar. D. Generational Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. ACM SIGPLAN Notices, 19(5), April 1984.

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

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