Code Sharing among Virtual Machines

Grzegorz Czajkowski[1], Laurent Daynès[1], and Nathaniel Nystrom[2]

[1] Sun Microsystem Laboratories,
2600 Casey Avenue, Mountain View CA 94043, USA,
Grzegorz.Czajkowski@sun.com, Laurent.Daynes@sun.com
[2] Computer Science Department,
Cornell University, Ithaca NY 14853, USA
nystrom@cs.cornell.edu

Abstract. Sharing of code among applications executing in separate virtual machines can lead to memory footprint reductions and to performance improvements. The design of a general and acceptable sharing mechanism is challenging because of several constraints: performance considerations, the possibility of dynamic class loading, dependencies between shared code and the runtime system, and the potential of adverse impact on the runtime's reliability and on ease of maintenance. This paper analyzes these tradeoffs in the context of two modifications to the Java^{TM} virtual machine (JVM^{TM}). The first allows for sharing of bytecodes and class information across multiple virtual machines, each of which executes in a separate operating system process, using shared memory. The second additionally enables the sharing of dynamically compiled code. Their design and performance are evaluated against two other approaches: running each application in a separate instance of an unmodified virtual machine, and running all applications in a single instance of a multitasking virtual machine.

1 Introduction

The idea of sharing executable code gained widespread acceptance in the mid-1980s, with the introduction of shared libraries [1, 9]. Shared libraries lower the system-wide memory footprint and enable faster application start-up. Providing support for shared libraries at the operating system (OS) level frees programmers from having to implement the sharing themselves. Today, shared libraries are an entrenched concept, available in optimized forms in most OSes. Due to the popularity of the Java^{TM} platform, it is common to come across a computer running many applications written in the Java^{TM} programming language [10] at any given time. One might ask whether in these settings the sharing of executable code across multiple virtual machines is as beneficial for the scalability of the JVM as shared libraries are for OSes.

In the context of the JVM, executable code includes the runtime representation of classes, methods and their bytecodes, and compiled code for methods generated by a just-in-time or dynamic compiler. Several characteristics of the

Java programming language and of the JVM make sharing executable code challenging. First, dynamic class loading makes the granularity of sharing small, at most a class. Second, the size and format of executable code evolve during program execution as the JVM identifies targets for optimizations and compiles them. Third, executable code is often intertwined with the runtime state of a program (e.g., pointers to objects subject to garbage collection can be embedded in executable code). Despite these difficulties, code sharing remains attractive because of its potential to decrease the memory footprint of virtual machines and to amortize costs related to on-demand class loading (e.g., parsing, verification, dynamic link resolution) and the architecture neutrality of class files (e.g., runtime quickening of interpreted code and runtime compilations), leading to both improved application execution time and to faster start-up time. This potential can be realized in differing degrees by sharing various forms of code (bytecode, quickened bytecode, compiled code, etc.)

This paper explores some of these issues by analyzing ShMVM. In ShMVM, each application is executed by a virtual machine running in a separate OS process. Executable code is shared among cooperating virtual machines using shared memory. Two versions of ShMVM have been designed and implemented: ShMVM-B allows for class information and method bytecode sharing while ShMVM-C additionally allows for compiled code sharing. This paper describes the design of ShMVM, highlighting the rationale for certain decisions. ShMVM was implemented by retrofitting an existing high-performance virtual machine with cross-process sharing capabilities; the discussion differentiates between issues inherent in the problem of sharing code and those due to the choice of the base virtual machine. The complexity of the internals of the chosen virtual machine swayed some design decisions towards minimizing the number of changes, and in several cases the outcome may be sub-optimal.

Two other architectures are used to evaluate the performance and robustness of ShMVM: (i) the currently standard way of executing multiple virtual machines, each in a separate process, without any sharing among them, and (ii) MVM [7], which transparently co-locates multiple applications in the same process. ShMVM and MVM are modifications of the same virtual machine. This enables meaningful quantitative discussion. We also analyze qualitatively the robustness of ShMVM. The use of shared memory may lead to degradation of robustness when compared to separate isolated JVMs.

This paper should be of interest to implementers of resource sharing virtual machines. In particular we conclude that while sharing code among processes has its advantages, the approach of MVM is better than ShMVM as a long-term solution to the effective use of resources by virtual machines. The rest of this paper is organized as follows. Sections 2 through 4 describe the design of ShMVM-B and ShMVM-C. Section 5 describes relevant code sharing details of MVM. Performance is the topic of Sec. 6. Robustness issues are the focus of Sec. 7. A general discussion is presented in Sec. 8. An overview of related work is given in Sec. 9.

2 Design Overview

Two versions of ShMVM have been implemented: ShMVM-B, which allows for sharing of class meta-data, including bytecodes, among virtual machines, and ShMVM-C, which additionally allows for sharing of compiled code. No distinction is made with respect to sharing between core (system) classes and application classes. The virtual machines participating in the sharing must be identical and must use the same version of the JDK classes. Both systems were implemented as modifications to the HotSpot Java^{TM} virtual machine [18] (referred to as HSVM from now on) version 1.3.1, client compiler, for the Solaris^{TM} Operating Environment executing on the SPARC^{TM} processor. Details specific to the two versions of ShMVM are described in the next two sections. Common design principles are discussed here.

In ShMVM each application is executed by a JVM in a separate OS process. The virtual machines cooperatively maintain a shared area that holds shared data. Whenever a virtual machine needs an item not found in the shared area, it computes the item and stores it there. The shared area is implemented as a memory mapped file. The first ShMVM process to map the file declares itself the primary and initializes the meta-data. After initialization, all processes are equally privileged to use the shared area. The shared area is mapped at the same virtual address by each participating JVM. This ensures that pointers to shared objects are valid across all JVMs. Often, shared data structures need to refer to data which are created as needed by each JVM and stored in their private area. Such shared-to-private references must encode a one-to-many mapping between one shared data structure and many private data structures, one per JVM. Direct pointers to the private area cannot be used for such references for two reasons. First, it would require each private data structure referenced from a shared object to be mapped at the same virtual address in all processes, which is impractical. Second, it prevents garbage collection to relocate private object referenced from a shared object. Our approach is to allocate an indirection table at a fixed location in the private space of each process.

Figure 1 shows how the address space of each process executing ShMVM is divided up into a private area, a shared area, and an indirection area. The last two must be at the same virtual address in all processes. The shared area holds objects that are shared across all processes. The private area contains all data private to a process, including the garbage-collected heap and thread stacks. Objects in any area can reference objects in the shared area directly, using their virtual memory addresses (e.g., pointers p_2 and p_4 holds the address of shared object o_2 in Fig. 1). Objects in the private area of a process can also directly reference objects in the private area of the same process (e.g., p_1). However, pointers to objects of the private area of any process (e.g., pointer p_{3a} in process A, or p_{3b} in process B) cannot be stored in the shared area, since they hold a virtual memory address that may not correspond to the same object in different processes. To solve this problem, each process maintains a private indirection table mapped at the same virtual address (i_1 in Fig. 1): objects in the shared area reference objects in the private area via an entry in the indirection table

(an indirection). Addresses to indirections (e.g., p_3 in Fig. 1) are valid across all processes, and therefore, can be stored in shared objects (e.g., o_1). Each indirections holds the virtual address of the object associated with it, which can be different for each process. For instance, shared object o_1 refers to indirection i, which has the same address p_3 in all processes; i holds the address of o_{3a} in process A and of o_{3b} in B.

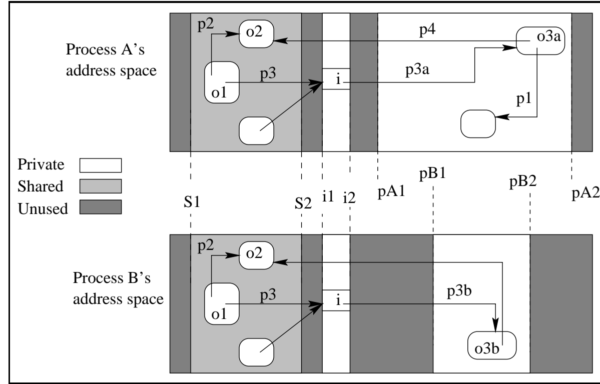

Fig. 1. The layout of memory areas in ShMVM.

Indirections are allocated as needed, when shared data that need to reference private ones are stored in the shared area. A field of a shared object that references a private object either holds a null value or the address to an indirection. JVMs initialize the entries of their indirection table to null. When a JVM uses a shared object for the first time, it initializes the indirections referenced from that object. Indirections may also be initialized lazily, in which case the null value is used to detect whether an indirection has been initialized. The garbage collector running in one process can relocate a private object referenced from the shared area independently of other processes by updating the indirections of its copy of the indirection table. This solution leverages virtual memory to efficiently support the one-to-many mapping between shared and private data.

Appropriate locks guard initialization, updates and look-ups in the shared area. A crash of one process holding a lock on a shared resource must not block forever other processes that are waiting for the lock to be freed. ShMVM's locking mechanisms relies on the atomic compare-and-swap instruction to implement non-blocking synchronization. To acquire a lock, the identifier of the locking process is atomically stored in the lock variable if the lock is not held by another process. The process will spin, yielding the processor, until the lock is available. Processes that terminated while holding a lock are detected by periodically asking for the process groups of all lock holders.

We assume that objects allocated in the shared area never become garbage, and thus are never collected. This assumption is roughly equivalent to preventing the unloading of shared classes and their compiled methods.

Finally, issues common to the design of both versions of ShMVM and of MVM are class initialization barriers and constant pool resolution barriers. A class initialization barrier tests whether a class has been initialized and triggers its initialization if it has not; in particular, the static initializer of this class is executed at this point. A constant pool resolution barrier tests whether a symbol in a class's constant pool has been resolved, and if not, proceeds to resolve it. Since both types of barriers always succeed except for the first time the barrier is encountered, JVM implementations commonly use dynamic code rewriting techniques, such as bytecode quickening [13] and native code patching to dynamically remove these barriers upon their first execution. Problems created by these barriers and respective solutions are described in the next three sections.

3 ShMVM-B Details

In ShMVM-B class information and method bytecodes are shared among virtual machines. The design revolves around the use in HSVM of a very infrequently collected area of the heap, called *permanent generation*, to store the runtime representation of classes, which includes descriptors for fields and methods, symbolic links, class constants and static variables, and method bytecodes. In ShMVM-B the permanent generation is split into a shared generation, shared by all JVM processes, and a private generation. Each process maps the shared generation at the same address. To minimize changes to garbage collection data structures, the shared generation is allocated contiguously to the private generation. Two kinds of objects are stored in the shared generation: (i) summaries of certain information about classes, and (ii) methods (including bytecodes).

3.1 Sharing Classes

In HSVM, the majority of the objects that collectively make up the runtime representation of a class are allocated in the garbage-collected heap. Each heap object starts with a header that includes a pointer to a klass object. The klass object understands the layout of a specific object type and knows how to reclaim it. To amortize the costs of class loading, we initially attempted to put as many objects of the runtime representation of a class (most notably klass, constant pool, and method objects) as possible in the shared generation. This approach presented several challenges: (i) klasses include a pointer to a C++ virtual function table located in private space, (ii) the resolved entries of the constant pool of a klass contain pointers to other klasses, to instances of `java.lang.String`, and to symbols (special objects used by HSVM to represent class symbols), and (iii) if klasses are shared, then the system dictionary, used to locate loaded classes, should be shared as well. For the reasons described below, in each case we decided against sharing.

The first problem is that each klass object contains a pointer to a C++ virtual method table (or *vtable*). Since the vtable is allocated in the process's data segment, it is in the private part of the address space. Indirectly accessing vtables via the indirection table would require changing the virtual method dispatch of C++, which is not realistic. One solution is to copy the vtable into the same area as the indirection table and to adjust the vtable pointer of klass objects upon their allocation. Unfortunately, the exact size of the vtable is difficult to compute without compiler support. Another, arguably clumsy solution would be to ensure that the vtables are loaded at the same addresses in all the virtual machines.

Sharing only the unresolved part of a constant pool requires changing its resolved entries so that they reference entries of the indirection table. However, the unresolved part amounts to only a fraction of the space occupied by the constant pool. So in effect, allocating one indirection per resolved constant pool entry is equivalent, in terms of space consumption, to replicate the constant pool for each process. The alternative is to share the whole constant pool and to directly reference resolved strings and symbols. This in turn requires pulling those strings and symbols in the shared area, as well as the table used to internalize them. The table of interned strings also records strings explicitly interned by applications via the `intern()` method of the `java.lang.String` class. This complicates sharing substantially, not the least by adding much more cross-process synchronization.

HSVM keeps track of all the klass objects in a system dictionary. If klass objects are stored in the shared area, the system dictionary needs to move there as well. The system dictionary contains private heap-allocated objects. If the system dictionary is shared, we must ensure that classes loaded by custom class loaders (i.e., not by the bootstrap or system class loaders) are isolated; allocating these classes in private space can accomplish this. Also, we would have to ensure that the protection domain of a class is not shared.

In addition to the difficulties it brings, sharing klass and constant pool objects would result in numerous cascading modifications to the existing run-time system, which goes against our goal of minimizing the changes to HSVM. The solution we finally opted for was to store klass summary objects in the shared area. Klass summary objects contain all information found in the class file as well as size information that is computed when a klass object is constructed. Klass information specific to an instance of the virtual machine, such as the class initialization state, and information that can be easily recomputed, are not stored in the summary object. The system dictionary is modified to look up summary objects in the shared generation. If a summary object is found, it is used to construct a klass object; otherwise, the runtime attempts to load the class file. To avoid class versioning problems, summary objects are only added to the shared heap for classes loaded with the bootstrap or system class loader.

The klass summary object contains a pointer to a copy of the class's constant pool. Unlike a klass object's constant pool, this copy does not have pointers to resolved classes or to instances of `String`, since link resolution should be performed at runtime by the individual virtual machines. The copy contains

pointers to symbols in the shared heap. Tables of interned strings and symbols are privately maintained by each process, and do not contain any of the symbol objects of the shared area. Because of this, in ShMVM, shared and private symbols are compared using their values instead of their addresses.

The klass summary also contains pointers to shared symbols for the names of the class's super class, interfaces, and source file. These symbols are handled in the same way as shared symbols pointed to by the constant pool. Other objects pointed to by klass objects, namely field description array, the inner class description array, and method descriptions, are shared with the klass summary object. The first two of these object types are immutable arrays of integers, so sharing them pose no problem. Sharing method descriptors is more complicated and will be discussed in Sec. 3.2.

Let us consider an object in a garbage-collected heap. This object has a klass pointer. Since all klasses reside in the private area of the JVM process, if the object is shared, its klass pointer must be indirected through the indirection table. If the object is private we do not need the extra indirection to access the klass object. However, we do not necessarily know statically if an object is shared or private. One alternative is to check if an object is shared when dereferencing the klass pointer. We could also do bounds checks on the shared space. The bounds of the shared space are compile-time constants, so the checks can be implemented without any additional loads. However, at least one branch would be required. Another solution would be to allocate an additional field in the object header that points to the private klass pointer. If the object is shared, the additional field points to the indirection table entry. If the object is private, the field points to the klass pointer field in the same object. This would increase the memory footprint of all heap-allocated objects, diluting one of the purposes of this work. Instead, we allocate an extra `self` field in each klass. This field contains a pointer back to the klass object itself. In each private object we store a pointer to its klass object. In each shared object, we instead store a pointer to the indirection table entry for the klass object adjusted by the offset of the `self` field in a klass object. Figure 2 shows the layout of these data structures. The instruction sequence for loading a klass object, identical for both shared and private objects, adds one load:

```
ld [this+4], klass_indirect
ld [klass_indirect + offset], klass
```

For private objects, the second indirection is redundant; thus, if it is known statically that an object is private, the second indirection can be eliminated. This is the case for all dynamically compiled code (not shared in ShMVM-B). Objects instantiated by the application (such as program data heap objects) are also never shared, and no additional cost is paid to obtain their klass pointers.

3.2 Sharing Methods

HSVM appends bytecodes at the end of objects representing methods. One option for sharing bytecode is to split the bytecodes out of the method object. This

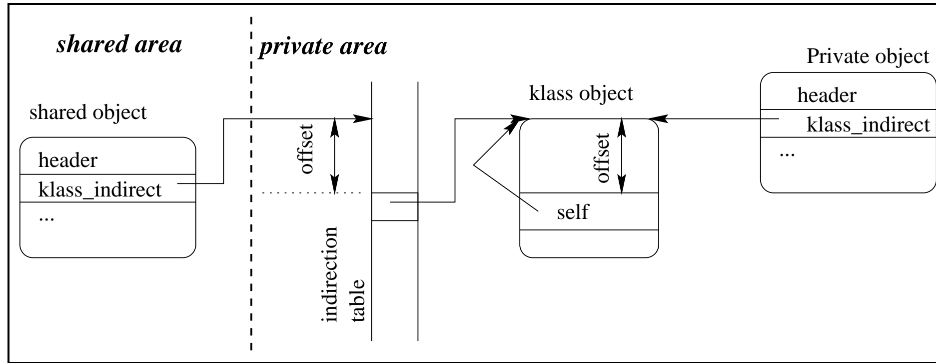

Fig. 2. Accessing klass objects in ShMVM.

required changing a considerable amount of code. Instead, we allocate the entire method object in the shared region. Other issues of interest include bytecode quickening, sharing invocation counters, and dealing with virtual method tables (*vtables*).

Bytecode Quickening. Bytecode rewriting, or quickening, is used to remove class initialization and constant pool resolution barriers. On its first execution, a given bytecode instruction executes the barrier operation, which sets up state for the corresponding quickened operation, and then overwrites the opcode of the instruction with that of another bytecode instruction; on subsequent executions of the fast instruction, the barrier operation is not performed. For constant pool resolution barriers, the state for the fast instruction is stored in a constant pool cache. Since bytecodes are shared, one virtual machine may quicken a bytecode and then another may execute the quickened bytecode without performing the barrier operation. This will cause the second virtual machine to crash since certain initialization actions are not performed. Disabling quickening completely ensures the barriers are always executed correctly. However, quickening is integrated tightly into the structure of the HSVM interpreter; the un-quickened versions of the bytecodes only perform the barrier operation while implementation of the actual bytecode operation is left to the quickened version of the bytecode. Our solution is to dynamically un-quicken bytecodes if the barrier operation has not been executed.

To simplify the implementation, quickening of the `new` bytecode and of the bytecode sequence `aload_0; getfield` was completely disabled. Unlike most of the other bytecodes, quickening of these bytecodes is not required for correct execution of HSVM. Quickening of the `ldc` bytecode for non-string constants need not be disabled at all. The slow version of these bytecodes is used solely to determine the type of the constant so that the correct load instruction is executed by the quickened bytecode. In HSVM, the slow version of `ldc` for string constants

may construct a `String` instance and install a pointer to it in the constant pool. In ShMVM-B the fast version of this bytecode also performs this operation.

The other bytecodes that may be quickened perform class constant pool resolution barriers or class initialization barriers. When executed, these bytecodes update the constant pool cache and then rewrite the bytecode. When a quickened bytecode is executed, we test if the constant pool cache for that bytecode is valid. If the cache is valid, the execution of the quickened bytecode can continue. Otherwise, the execution is dispatched to the un-quickened version of the bytecode. The bytecode is not re-written with the un-quickened version since this may result in a race condition. The code for the un-quickened bytecode will rewrite the bytecode with the fast bytecode, but this is safe since the same quickened opcode will be written. Testing the constant pool cache entry validity requires only a null pointer check.

Sharing of Invocation Counters. HSVM uses method invocation counters to determine which methods are frequently invoked. A method is compiled when its counter reaches a certain threshold value. In ShMVM-B the invocation counter of a method may be shared (default) or private, depending on a runtime flag.

If counters are shared, methods "hot" in one application may cause compilation of the method by another virtual machine. Hopefully, such methods will be hot in other applications as well. Virtual machines that start later will compile hot methods earlier than they would have with private invocation counters. Less time will be spent executing interpreted code while the invocation counter rises toward the compilation threshold. However, compilation may occur in several virtual machines that execute a given method even if in an individual application the method would not have reached the compilation threshold. This compilation may decrease the aggregate throughput. If the shared region exists for a long enough time, a new virtual machine attaching to it may behave as if all of the methods it executes required compilation on first invocation.

Private invocation counters are implemented by creating a one-element array in private space and a pointer to it through the indirection table in the method description object. This requires two extra indirections to access the counter compared to the path length of accessing shared counters. Since only the interpreter uses the counters, the overhead of these extra loads is negligible.

Method Vtable Index. Klass object embeds a vtable that contains pointers to method descriptors. Each descriptor includes the index to the vtable entry where its pointer is stored. Because method descriptors are shared, vtable indexes must be valid for all virtual machines.

In HSVM, the vtable is ordered by comparing addresses of the symbols holding each method name. Symbols are allocated in the permanent generation and their relative order remains fixed. However, two processes may allocate symbols at different addresses. The vtable sorting implementation was consequently changed in ShMVM to do string comparisons on method names.

4 Details on ShMVM-C

ShMVM-C extends ShMVM-B with the ability to share dynamically compiled code. In particular, a virtual machine can execute the native code of a method already compiled by another virtual machine, without having to interpret the method at all. In HSVM compiled methods (called nmethods) are stored in a contiguous area of memory called the code cache. In ShMVM-C this area is mapped at the same address in all virtual machines. Since only method description objects (already located in the shared area – Sec. 3.2) contain references to their corresponding nmethods, no new infrastructure for finding nmethods had to be implemented. In particular, whereas ShMVM-B needs an indirection from a method object to its nmethod, ShMVM-C can use a direct pointer, like HSVM.

Sharing the code cache raises several issues. First, the code cache refers to many data structures dynamically allocated outside of the code cache or the garbage collected heap. Examples of such data structures include small fragments of code, maps of object references, and caches of exception handler locations. These data structures must be accessible to all processes using the code cache and were therefore moved into the code cache for simplicity. References in these data structures to objects stored in the garbage collected were indirected.

Second, the dynamic compiler embeds pointers to objects allocated in the garbage-collected heap in the code that it emits. This includes pointers to klasses, methods, instances of `java.lang.String` and `java.lang.Class`, and objects implementing inline caches. In ShMVM-C, all such embedded pointers must refer to locations shared between processes in order for the compiled code to be sharable.

Method objects are already in the shared generation, and thus can be left embedded in compiled code. Inline cache objects serve an optimization that significantly speeds up the dispatch of virtual methods. For both simplicity and efficiency, they were moved into the shared generation so that they can be referenced directly. Klass objects, for reasons already explained in Sec. 3.1, are not shared, and therefore the indirection mechanism (Sec. 2) is used to access them. That is, pointers to the indirection table entries for instances of private klass objects are embedded in the compiled code. The generated code was modified to properly handle the double dereferencing to get to klass objects. Whenever a virtual machine executes a method for the first time and that method has already been compiled by another virtual machine, the first dereference (following a pointer embedded in the compiled code) will return a null value. In this case, the code is dispatched to a routine that finds the corresponding method and properly initializes the entry in the indirection table.

The only instances of `java.lang.String` for which a reference is embedded in compiled code are resolved strings from the constant pool of the class of the compiled method. For each such string, the corresponding `ldc` bytecode instruction contains the constant pool index for this string. As for classes, pointers to these strings are replaced with pointers to entries in the indirection table. After each dereference of such a pointer, the compiler inserts a null pointer test that

branches to an upcall to the runtime. The upcall specifies the index of the constant pool where the string may be found. Thus, when a process dereferences this embedded pointer for the first time the upcall to the runtime will find the corresponding local string and update the indirection table entry accordingly before resuming execution of the compiled code. With these changes, compiled code does not contain any direct pointers to virtual memory areas that are private to one process, making it sharable across virtual machines.

Another issue is related to class initialization barriers. Each klass whose pointer can be embedded in compiled code is associated with two consecutive entries in the indirection table. Each of these entries can contain either a null value or the pointer to the klass object private to the current process. The first entry corresponds to the loaded status (the null value indicates that the class has not been loaded yet) and the second one to the initialized status (the null value indicates that the class has not been initialized yet). Upon loading of a class the "loaded" entry is updated with the klass pointer, but its "initialized" entry is left set to the null value. Compiled code must test the value of the second entry at places that may trigger the initialization of a class, i.e., in code generated for `new`, `getstatic`, `putstatic`, and `invokestatic` bytecode instructions. Using this organization, access to a class at points requiring test for class initialization adds only two instructions to the original sequence generated by HSVM:

```
sethi hi(ind_tbl_entry_addr), entry
ld [entry + lo(ind_tbl_entry_addr)], k
brz,k,a class_initialization_stub
ld [entry+4], k
```

The original pair of `sethi`/`add` instructions is replaced with a pair `sethi`/`ld` to embed a pointer to an entry in the indirection table, instead of a pointer to a klass object. A branch on a register value tests the result of the introduced load added to obtain the klass pointer from the indirection. When the class is not already initialized, the load in the annulled delay slot of the branch instruction stores the contents of the second entry in the register, and control is transferred to a stub that invokes the runtime class initialization routine.

5 Code Sharing in MVM

MVM [7] is an implementation of the JVM capable of executing many programs simultaneously within a single OS process. MVM supports all the features and APIs of the Java platform. Multiple invocations of the standard invocation API within a single process actually result in the creation of multiple instances of the JVM within that process, each capable of executing a program. Each JVM instance, referred as a task hereafter, is a set of data structures that captures the part of the execution context of a program that cannot be shared (e.g., static variables, class initialization status, etc.). The aggressive sharing of the JVM data structures and of the runtime representation of classes, including dynamically compiled code, contributes to making the size of a program execution

context small. Privileged programs can create and control tasks using a preliminary version of the application isolation API [12].

For the purpose of comparison with the other approaches to code sharing explored in this paper, only the changes to the runtime representation of classes, to the interpreter, and to the dynamic compiler are described in what follows.

5.1 Shared Class Runtime Representation

Sharing heap-allocated runtime representations of classes is problematic in ShMVM because each program executes in different address space. MVM does not suffer from this since all tasks execute in the same address space and have access to the heap. Therefore, sharing data accross tasks does not require using a different pointer format or removing objects from under the contol of the garbage collector. As a result, MVM barely changes the runtime representation of classes used in HSVM.

Most of the runtime representation of a class is already independent of any particular execution context and can therefore be shared as is. This sharable portion includes the constant pool, debugging information, the descriptions of methods and fields, including information resolved at runtime such as the offset of an instance variable from the beginning of an object or the index of a method in a virtual table, and, given appropriate changes to the interpreter, the bytecode of methods. The runtime constant pool cache (a subset of the runtime constant pool optimized for use by both the interpreter and code produced by the runtime compiler) can also be shared after a few minor modifications.

Data that cannot be shared across program execution contexts, that is, the task-dependent data, are relatively small, and include the storage for static variables, the objects that constitute the program-visible representation of classes, such as instances of `java.lang.Class` and other relevant objects (e.g., class loader, signers, etc.), and data describing the initialization state of the class. In HSVM, all of the above is either embedded in, or referenced from, a single heap-allocated klass object (Sec. 3.1).

MVM replaces the task-dependent data of each klass object with a single reference to an array of references to taskKlassMirror objects (TKMs). Each TKM encapsulates the task-dependent part of a class for a given task. Both TKM tables and TKMs are heap-allocated. TKM tables are sized to correspond to the maximum number of tasks supported. Tasks are uniquely identified within the virtual machine using an index to a task table that keeps track of ongoing tasks. Each program thread is tagged with the unique identifier of its task and always runs in the context of the same task. Obtaining the TKM corresponding to a given task is a matter of indexing the TKM table of the corresponding klass with the current task's identifier stored in the current thread.

5.2 Task-reentrant Initialization Barriers

Class initialization barriers cannot be entirely eliminated in MVM because both bytecodes and the code generated by the dynamic compiler are shared by mul-

tiple tasks, which may each be at different stages of initialization for a given class. Testing whether a task has initialized a class amounts to checking whether the entry in the class's task table for that task is non-null. The testing part of the class initialization barrier takes three instructions: loading a unique internal task identifier from the current thread data structure, loading the address of the corresponding TKM from the class's task table (indexed by the task identifier), and then branching to the appropriate class initialization code if the returned address is null.

Both the initialization status of a class and the thread initializing it are kept in the TKM. The main issue is to locate the TKM corresponding to the initializing task when one of its threads is dispatched to the runtime to initialize the class. The entry in the TKM table cannot be used to store the TKM 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. A scheme similar to the one described in Sec. 4 addresses this issue: the TKM table holds two references to the same TKM for each task. Each reference is set up at a different time: the first one is set up during class loading, and the second one once the class is fully initialized. Class initialization barriers test the second entry only: when the test performed by the barrier fails, the TKM for the current task can be simply obtained from the first entry. The first entry is also useful for accessing the static variables of a class for a given task when the task is not fully initialized (e.g., an access to the static variable may be required by the thread initializing the class while executing a static initializer of the class).

5.3 Bytecode Interpretation

MVM leaves the interpretation of all standard bytecodes unchanged. Modifications are required only for the interpretation of some of the quick versions of 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.

As explained in Sec. 5.2, 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 additional load instruction before the barrier code described in Sec. 5.2 in order to fetch the TKM table of the class. The net increase in the path-length of these bytecodes is thus 4 instructions. The quickened versions of `getstatic` and `putstatic` need, in addition to the class initialization barrier, the TKM 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 TKM of a class. Thus, once the barrier is passed, the static variable can be obtained with a single memory load. This only adds 3 instructions to the path-length of `getstatic` and `putstatic` because the constant pool cache entries for these instructions have been modified to store a reference to a class's TKM table instead of a reference to a klass.

5.4 Sharing Compiled Code

Because MVM executes all programs in the same address space, it did not introduce many of the compiler-related problems that were encountered during the design of ShMVM-C. In particular, no changes were necessary to deal with embedded pointers (see Sec. 4). The code originally emitted by HSVM's compiler can almost be shared as is between tasks. The only aspect that needed care is class initialization barriers.

MVM addresses this problem by augmenting the compiler with a set of simple optimizations that determine, independently of any runtime state, when a class initialization barrier is necessary, and generate a task re-entrant class initialization barrier if so. In particular, class initialization barriers are omitted if their target is one of (i) the class defining the method being compiled, (ii) a super-class of the above, (iii) a class initialized at virtual machine startup whose initialization was not triggered by the method being compiled, or (iv) a class for which a barrier has been already emitted upward the instruction stream of the method being compiled. Because the dynamic compiler now generates code that does not make any assumptions about the initialization state of classes, 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.

Two additional modifications to the compiler make the compiled code task re-entrant. First, code generated to access static variables is modified to obtain the reference to the TKM corresponding to the current task, where the static variables are stored. The reference to the TKM is either obtained via a class initialization barrier (if present), or more directly, by indexing the class's TKM table with the current task's identifier. Second, code generated to enter and to exit a class's monitor is modified to locate the appropriate instance of `java.lang.Class`.

6 Performance

To gain insight into the performance of ShMVM, we measured its start-up time, the performance of the SPEC JVM98 benchmarks [17], and the footprint savings. The experimental setup consists of a Sun Enterprise^{TM} server with four UltraSPARC^{TM} II processors, with 4GB of main memory, running the Solaris Operating Environment, version 8. The code base of HSVM (Sec. 2) is used as the implementation basis for ShMVM and for MVM.

6.1 Start-up Latency

We define start-up latency as the time elapsed between issuing the java command and the moment when the main method of the class specified in command line argument begins execution. For HSVM this was measured by recording the total execution time required to execute *JustReturn*, an application that has only the

return statement in its main method. This captures the process startup time and the bootstrapping of the virtual machine present in each execution. The same approach was used to measure the start-up latency of ShMVM. The typical use of ShMVM is as a number of virtual machines collectively maintaining a shared area; since only the first of them will initialize bootstrap-related information in the shared area, the start-up latency reported here is for a subsequent execution of ShMVM. For MVM the following strategy was used: a simple application manager, which executes as a task, listens on a socket. Whenever a request arrives, the manager immediately starts a new task to execute *JustReturn* and replies with an "ok" message upon termination of that task. The time elapsing between sending the request and receiving the reply is reported as MVM's start-up time. This reflects the intended use of MVM as a multi-tasking virtual machine, in contrast to the HSVM and ShMVM models, in which JVM OS processes are started for each new application.

The results relative to the start-up time of HSVM are presented in Fig. 3. The modest decrease gained in ShMVM is a result of a faster bootstrap sequence, which does not require loading system classes shared in this architecture. However, the bulk of the startup latency is related to starting an OS process and initializing the runtime. In MVM these issues are not present in the start-up of a task. As a consequence, the start-up latency result is only 2.7% of the time necessary to start up HSVM.

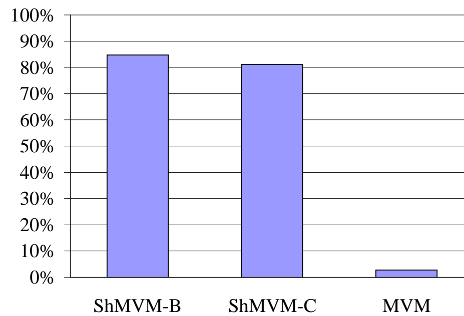

Fig. 3. Startup latency of ShMVM-B, ShMVM-C, and MVM relative to HSVM.

6.2 Application Performance

Total application execution time is another measure of performance impact. The same experimental strategy as with start-up latency measurements was used to execute SPEC JVM98 benchmarks. The execution time may differ depending on how many times a given benchmark has been executed before.

For ShMVM-B, the first execution will typically be longer, as it needs to compute the data stored in the shared area. Since no new classes are loaded

after the first execution of the benchmark, the execution time of the second and subsequent executions (instances) of the benchmark are the same. For ShMVM-C, the execution time of any instance of the benchmark can be faster than the previous one as more compiled methods may become available. Similar effects can be observed for MVM for the same reasons.

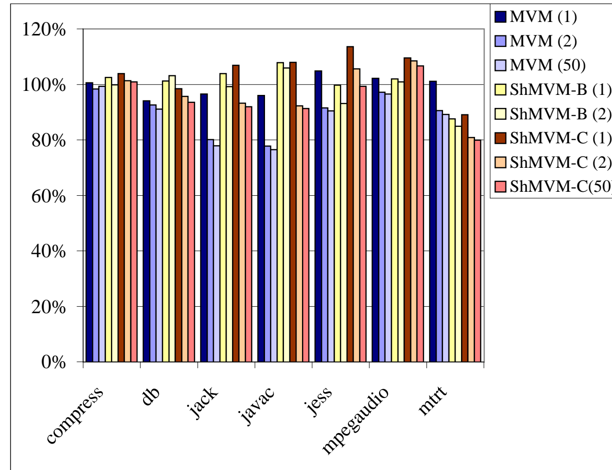

Fig. 4. Performance of ShMVM-B, ShMVM-C, and MVM relative to HSVM.

We ran both versions of ShMVM with shared and with private invocation counters (Sec. 3.2). We have not noticed any significant differences in the execution time between these two schemes, which implies that for these programs the effect of shared invocation counters is negligible. Results are reported on Fig. 4. For ShMVM and for MVM the numbers in parentheses indicate which instance of the benchmark was measured. Several observations can be made. First, for both MVM and ShMVM-C the difference in execution time between the second and the fiftieth instance of any of the benchmarks is much smaller than the difference between the first and the second instance. This indicates that all the “hot” methods are compiled during the first execution and the subsequent compilations of “colder” methods do not impact performance for these programs. The situation may be different for other programs, with more dynamic and less predictable class loading and method execution behavior. Second, MVM is faster than any of the versions of ShMVM for all but one benchmark. The reason for mtrt behaving this way is not clear. In the case of ShMVM-B the slowdown relative to MVM is understandable, since it does not enjoy the benefits of shared compiled code. Shared-to-private references planted in the generated code make ShMVM-C slower than MVM. In particular, two such references are followed on a non-static method invocation.

These measurements indicate that the costs of handling shared-to-private references outweigh the benefits of sharing compiled code. Finally, when compared to HSVM, the first execution of a benchmark under MVM executes on the average in the same time as with HSVM, while ShMVM-B and ShMVM-C are typically slower than HSVM. This is due to the cost of storing data in the shared area.

6.3 Memory Footprint

The sizes of shared areas are shown in Table 1. The first column contains the size of the shared generation in ShMVM-B after the execution of the first instance of a benchmark. This does not change after later executions for the same benchmark, since for the SPEC JVM98 programs the set of loaded classes is always the same. The next three columns contain, respectively: the size of the code cache after the execution of the first, second, and fiftieth execution of the benchmark in ShMVM-C (to get the total size of shared code in ShMVM-C, these values must be added to the first column, as ShMVM-C is an extension of ShMVM-B).

The memory savings due to sharing bytecodes are uniform across the benchmark programs, in the range of 300-500KB. For ShMVM-C, the size of the code cache varies much more across the benchmark programs. This indicates that the size of loaded bytecodes and the size of compiled "hot" methods are not strictly correlated. Moreover, in some scenarios the size of data in the shared generation is larger than the total size of compiled code but in some others is smaller, which precludes giving a clear verdict on what is more important to share from a footprint standpoint: bytecode or compiled code.

Table 1. Size (in KB) of the shared generation and code cache in ShMVM.

	ShMVM-B	ShMVM-C (1)	ShMVM-C (2)	ShMVM-C (50)
compress	297	96	96	96
db	304	126	137	686
jack	363	858	859	921
javac	492	1487	1500	1629
jess	392	409	415	582
mpegaudio	364	510	511	577
mtrt	340	396	434	467
JustReturn	266	42	42	42

The savings reported in Table 1 are effective when at least two instance of the same program execute simultaneously. However, bootstrap classes are necessary for executing any application. The last row reports their size (the data was obtained by running *JustReturn*). Bootstrapping stores about 266KB-worth of methods and klass summaries in the shared generation, and a further 42KB of compiled methods in the code cache. These savings are applicable to any

application executed in ShMVM. For applications using large sets of classes, such as the Swing package, the savings would be much bigger.

It is interesting to compare these numbers and the memory footprint of a virtual machine process. For instance, the pmap utility reports that for HSVM running *JustReturn*, the size of resident memory pages due to C/C++ libraries (both system libraries, such as libc and libCrun, and JVM-specific ones) is 5.06MB, out of which 4.14MB are attributed to shared pages. This means that starting a new instance of ShMVM (either ShMVM-B or ShMVM-C) incurs almost 1MB of memory footprint due to non-shareable (private) segments of libraries necessary for virtual machine's startup. Thus, unless an application generates a large amount of code, the savings realized by sharing in ShMVM are smaller than the footprint of a new process. This should be contrasted with MVM, where, in addition to sharing class information (including bytecode) and compiled code, private segments of libraries are shared as well.

7 Robustness

ShMVM sacrifices robustness for scalability when compared to executing each application in its own JVM OS process with no sharing among the JVMs. First, errant native code in any of the virtual machines may write over the shared area, corrupting it and potentially causing other virtual machine processes to crash. This problem can be somewhat mitigated by write-protecting the shared area between writes to it. However, this solution only decreases the window of opportunity for the problem occurring at the expense of potentially high performance overhead. Weakened robustness is mostly a result of having a writeable shared area, but on the other hand the largest win in memory footprint reduction comes from sharing things that are writeable, e.g., the resolved constant pool, compiled code, the system dictionary, etc. The largest gain in terms of runtime cost results from eliminating compilation and interpretation by sharing compiled code. However, in HSVM as well as in ShMVM, compiled code evolves at runtime (inline caches, safe-point traps and fix-up of embedded pointers at garbage collection time are the main examples of this; recompilations and deoptimizations are other, more elaborate possible cases). Frequent code rewriting is very likely to make the use of virtual memory protection in order to improve robustness too expensive.

Several approaches may be used to address this problem. First, read-only meta-data images can be generated and then memory-mapped by virtual machines. For instance, desktop applications using a large number of GUI components can benefit from such a solution as all the Swing classes can be put into the shared area. Another approach would be to store meta-data in a server process and access it via copying inter-process communication. This will not decrease the footprint, but performance may improve. A more difficult problem is surviving virtual machine crashes when at the point of crash a shared resource was accessed. While we can detect and deal with locks on shared resources being held during crashes, a limitation of our locking scheme is that a process holding

a lock can die, leaving a shared data structure in an inconsistent state. Using recoverable memory, such as RVM [15] or Rio Vista [14], can address this problem. Recoverable memory provides atomic updates and persistence for a region of virtual memory, and allows programs to manipulate permanent data structures safely in their native, in-memory form. A simple, lightweight layer that handles atomicity and persistence is also provided. These transactional guarantees simplify programming by restricting the number of states in which a crash can leave the system. The feasibility of implementing shared meta-data areas for ShMVM with a low-overhead recoverable memory system, depends on what is actually shared.

Yet another alternative would be to encode shared meta-data in shared libraries. Immutable parts of shared libraries can be used to store pre-quickened bytecodes, while the mutable parts, lazily turned from shared to private by the copy-on-write mechanism, would store resolved data forming the constant pool cache. Some of the properties of the shared library mechanism and the optimized support provided by modern OSes makes this approach promising, especially for class meta-data. Sharing compiled code may be more problematic, especially in the presence of code rewriting.

8 Discussion

Even though improvements are certainly possible to our design and implementation, we have enough data for a general summary of the opportunities and broadly defined costs associated with designs similar to ShMVM. In the following discussion, properties of ShMVM are contrasted with those of MVM, since both systems aim at improving the resource utilization of the JVM.

Let us first look at the effort associated with modifying a high-performance, industrial-strength virtual machine. Less design and implementation effort was required for MVM than for ShMVM. Splitting klass data structures in MVM was greatly simplified by not having to deal with different address space issues, and by leaving the resulting data structures in the garbage-collected heap. This led to virtually no modifications to the garbage collector. Also, because of the single address space assumption, practically the entire runtime representation of a class was shared and sharing of compiled code was simpler. For instance, no changes were necessary to the inline cache optimization, and embedding of pointers into compiled code was left unchanged except for places requiring access to static variables and to class initialization barriers. In these cases, embedded pointers to klass objects were simply replaced with embedded pointers to TKMs. Dealing with class initialization barriers in MVM was relatively simple: the only difficulty was augmenting the compiler with an analysis to eliminate unnecessary barriers. Implementing the other parts of MVM, such as fast paths for class loading, linking, and initializing, and accessing mutable parts of classes was much simpler than the corresponding changes for ShMVM.

The use of multiple address spaces has been the main source of problem in ShMVM. Code sharing was particularly challenging. In ShMVM splitting klasses

proved much more difficult because of data structures located in the garbage-collected heap in HSVM that had to be moved to the shared area in ShMVM. This resulted in substantial changes to both the garbage collector and the dynamic compiler. Although bytecodes are shared, most of the runtime representation of a class had to be replicated, as well symbol tables, system dictionary etc. As a consequence, other issues, such as the computation of virtual method tables, emerged and had to be addressed.

Choosing MVM or ShMVM shifts the optimization effort to different parts of the virtual machine. For instance, in ShMVM the main issue is handling shared-to-private pointers. Optimizing for this and minimizing the number of indirect references can significantly impact the design of klass objects, of inline caches, of the code executed to interpret quickened bytecodes, and of the mechanism to detect whether a class initialization barrier can be removed. Only the last of these issues is present in the design of MVM. However, MVM virtualizes non-constant parts of classes, and optimizing their access path is important. The differences in design and optimization focuses are likely to become even more pronounced when rolling both systems forward to target a more sophisticated compiler. For instance, a larger number of direct references, such as references to Java objects on the heap, can be embedded in the compiled code. Optimizing ShMVM is challenging in this case. Certain optimizations, beneficial in the case of a single non-shared address space, are difficult to deal with in the presence of both the shared and private areas, and the approach of ShMVM may preclude some of these optimizations. Both MVM and ShMVM would be impacted by more aggressive virtual method inlining and by de-optimizing when an optimistic assumption, such as assuming a class has only one subclass, fails.

MVM is better than either version of ShMVM on all performance-related metrics (more data on MVM can be found in [7]). It is also worth pointing out that ShMVM currently does not have any scheme to change protection of the shared area's pages to increase robustness, which would have a negative impact on its performance, especially for ShMVM-C. However, the following three points must also be weighed in. First, MVM is just another OS process, with a single set of permissions for accessing the file system, etc. This is not an issue for the construction of MVM-based Web servers, application servers, and for single-user desktop scenarios, but the construction of multi-user environments is more challenging. In ShMVM, each virtual machine can have its own set of permissions, which facilitates the handling of multi-user requirements. Second, in MVM user-supplied native code transparently executes in a separate process, which may adversely impact performance. In ShMVM this may also be an issue since errant native code can clobber the shared area. Protecting and un-protecting these areas may be expensive if the shared data is frequently updated. Third, robustness is an issue in the presence of virtual machine bugs: a bug in the runtime system can corrupt or crash all the tasks co-located in MVM. This is also the issue for ShMVM although it has a lower likelihood that the effects of such a bug are not isolated to one application only.

Achieving as much sharing as accomplished with MVM but with cross-address space sharing â la ShMVM requires a great deal of changes to the underlying virtual machine; so much in fact that a from-scratch re-design looks like an attractive option. MVM was easier to engineer, and required fewer changes for better results than ShMVM. Following the ShMVM model may quickly turn into a slippery slope: when one item is shared, it is tempting to share objects pointed to by that item. No matter how the graph of references among runtime data structures is cut by the boundaries of shared and private address spaces, the issue of dealing with references and dependencies spanning these spaces is difficult, especially given the complexity of modern high-performance virtual machines. Minimizing the number of such inter-space references is important for performance, for implementation complexity, and for robustness (if the shared area is writeable). Our experience suggests that removing inter-space references and co-locating all of them in a single address space in a MVM-like style is the most attractive long-term solution.

9 Related Work

Quite a number of projects have aimed at conserving resource consumption of the JVM. The majority of these efforts focus on co-locating applications in the same virtual machine. Detailed overview of these efforts can be found, for instance, in [2, 3, 5, 6, 11].

The only account of work similar to ours we have found is [8], which describes IBM's implementation of the JVM for OS/390. This system, aimed at server applications, is interesting in several respects. Multiple JVMs can share system data (e.g., classes, method tables, constant pools, etc) stored in a shared memory region, called the shared heap. The shared heap is designed to store system data but can also store application data that can be reused across multiple instances of the JVM. The shared heap is never garbage collected, and cannot be expanded. The JVMs use the shared heap to load, link, and verify classes. A JVM need not perform any of these actions for any class that has been loaded by another JVM; this includes the bootstrap and system classes. A common class loader is used to share name spaces across a set of JVMs. Compiled code is not shared. The account presented in [8] briefly discusses these issues at a high level, without expounding on challenges and alternatives; it also does not discuss the performance of the system. It would be very interesting to compare the design and problems faced in that work with the issues we had in building ShMVM. This system presents another interesting approach to conserving resources: each JVM is executed in a large outer loop, which accepts requests to execute programs. After a program has been executed and it is determined that it has not left any residual resources behind (e.g., threads, open files, etc), the JVM can be immediately used to execute another request. Thus, multiple virtual machines can concurrently share resources through the shared heap, but additionally, each of them reduces start-up latency via sequential execution of applications.

A similar system is described in [4]. Performance data presented there are promising from the perspective of reduced start-up time, but monitoring and managing the transition to "clean slate" virtual machine can be a challenging task. Compiled code is shared in [4], but no account is given of the complexity and benefits of this feature in their system.

The Quicksilver quasi-static compiler [16] aims at removing most of the costs of compiling bytecodes. Pre-compiled code images of methods are generated off-line. During loading they need to be "stitched", that is, incorporated into the virtual machine using relocation information generated during the compilation. Stitching removes the need for an extra level of indirection since relevant offsets in stitched code are replaced with the actual addresses of data structures in the virtual machine. These design decisions form an interesting comparison with ShMVM, where code compiled on-line is shared in order to lower memory footprint and where the sharing of compiled code introduces an extra level of indirection in certain cases. The approach of Quicksilver may also be complementary to ShMVM and to MVM: certain meta-data can be computed off-line and used to pre-populate shared areas.

10 Summary

This paper discusses the design, selected implementation details, and performance of ShMVM, an architecture that allows Java virtual machines to share executable code. The implementation is based on an existing high-performance virtual machine. Partitioning the runtime data structures across shared (located in shared memory) and private heaps creates numerous problems when introduced into a complex, well-tuned runtime designed with an implicit, inherent, and pervasive assumption of a single, uniform, non-shared address space. After addressing these issues and after the evaluation of our results we conclude that JVM architectures promoting multitasking in a single process are much more attractive as a long-term approach to improving the scalability of the virtual machine. This is, in a way, a negative result of significant value, as various groups have recently contemplated improving the efficiency of their virtual machines by designs similar to ShMVM. Hopefully our findings will lead to faster, leaner, and more robust future virtual machine architectures.

Acknowledgements. The authors are grateful to Dave Dice, Misha Dmitriev, Mick Jordan, Fred Oliver, Glenn Skinner, Pete Soper and Mario Wolczko for their comments and suggestions.

Trademarks. Sun, Sun Microsystems, Inc., Java, JVM, 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.

References

1. Arnold, J.: Shared Libraries on UNIX System V. Summer USENIX Conference (1986), Atlanta, GA.
2. Back, G., Hsieh, W., Lepreau, J.: Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. 4th Symposium on Operating Systems Design and Implementation (2000), San Diego, CA.
3. Balfanz, D., Gong, L.: Experience with Secure Multi-Processing in Java. Technical Report 560-97, Department of Computer Science, Princeton University (1997).
4. Borman, S., Paice, S., Webster, M., Trotter, M., McGuire, R., Stevens, A., Hutchinson, B., Berry, R.: A Serially Reusable Java Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing. Technical Report TR29.4306, IBM Corporation.
5. Bryce, C., Vitek, J.: The JavaSeal Mobile Agent Kernel. 3rd International Symposium on Mobile Agents (1999), Palm Springs, CA.
6. Czajkowski, G.: Application Isolation in the Java Virtual Machine. ACM OOPSLA (2000), Minneapolis, MN.
7. Czajkowski, G., Daynès, L.: Multitasking without Compromise: A Virtual Machine Evolution. ACM OOPSLA (2001), Tampa, FL.
8. Dillenberger, W., Bordawekar, R., Clark, C., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M., Samuel, F., St. John, R.: Building a Java Virtual Machine for Server Applications: The JVM on OS/390. IBM Systems Journal, Vol. 39, No 1, 2000.
9. Gingell, R., Lee, M., Dang, X., Weeks, M.: Shared Libraries in SunOS. Summer USENIX Conference (1987), Phoenix, AZ.
10. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. 2nd edn. Addison-Wesley (2000).
11. Hawblitzel, C., Chang, C-C., Czajkowski, G., Hu, D., von Eicken, T.: Implementing Multiple Protection Domains in Java. USENIX Annual Conference (1998), New Orleans, LA.
12. Java Community Process.: JSR 121: Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp.
13. Linholm, T., Yellin, F.: The Java Virtual Machine Specification. 1st edn. Addison-Wesley (1996).
14. Lowell, D., Chen, P.: Free Transactions with Rio Vista. 16th ACM Symposium on Operating Systems Principles (1997), Saint-Malo, France.
15. Satyanarayanan, M., Mashburn, H., Kumar, P, Steere, D., Kistler, J.: Lightweight Recoverable Virtual Memory. 14th ACM Symposium on Operating Systems Principles (1993), Asheville, NC.
16. Serrano, M., Bordawekar, R., Midkiff, S., Gupta, M.: Quicksilver: A Quasi-Static Compiler for Java. ACM OOPSLA (2000), Minneapolis, MN.
17. Standard Performance Evaluation Corporation.: SPEC Java Virtual Machine Benchmark Suite (1998). http://www.spec.org/osg/jvm98.
18. Sun Microsystems, Inc.: Java HotSpot^{TM} Technology. http://java.sun.com/products/hotspot.