# Sharing the Runtime Representation of Classes across Class Loaders

Laurent Daynès and Grzegorz Czajkowski

Sun Microsystems Laboratories
Laurent.Daynes@sun.com Grzegorz.Czajkowski@sun.com

**Abstract.** One of the most distinctive features of the Java$^{TM}$ programming language is the ability to specify class loading policies. Despite the popularity of class loaders, little has been done to reduce the cost associated with defining the same class by multiple loaders. In particular, implementations of the Java virtual machine (JVM$^{TM}$) create a complete runtime representation of each class regardless of how many class loaders already define the same class. This lack of sharing leads to poor memory utilization and to replicated run-time work. Recent efforts achieve some degree of sharing only when dynamic binding behaves predictably across loaders. This limits sharing to class loaders whose behavior is fully controlled by the JVM. As a result applications that implement their own class loading policies cannot enjoy the benefit of sharing.
We present a novel technique for sharing the runtime representation of classes (including bytecodes and, under some conditions, compiled code) across arbitrary user-defined class loaders. We describe how our approach is applied to the multi-tasking virtual machine (MVM). The new multi-tasking virtual machine retains the fast start-up time of the original MVM while extending the scope of footprint savings to applications that exploit user-defined class loaders.

## 1 Introduction

One of the most distinctive features of the Java$^{TM}$ programming language [10] is the ability to define class loading policies [13]. Class loaders are exploited in a wide range of applications, such as scripting environments, IDEs, runtime code injection tools, aspect-oriented programming platforms, web browsers, servlet engines, and application servers.

Class loaders are popular for several reasons. They provide separate namespaces, which allows a program to link components regardless of whether they include different versions of the same classes or different classes with the same name. This feature enables the implementation of a form of isolation, where carefully crafted software modules can be loaded multiple times without interfering with one another [4]. Class loaders also give programs control over the location where classes are loaded from, and an opportunity to transparently enhance third-party code by providing mechanisms for its interception and modification (via bytecode transformations) before it is linked with the rest of the managed execution environment.

Using class loaders comes at a cost, however. JVM implementations typically replicate the entire runtime representation of a class in memory for each class loader that defines the class[1]. Defining a class multiple times also replicates the effort to create an optimized runtime representation, by repeating class file parsing, construction of main-memory data structures, bytecode verification, (optional) bytecodes quickening, resolution of constants, and identification and recompilation of frequently used methods.

The use of delegation relationships between class loaders, where one class loader can delegate the definition of a class to another, helps to limit these problems but can rapidly become error prone as the complexity of the delegation relationships increases. Besides, not all uses of class loaders can be accommodated with delegation. For example, delegation is inadequate when multiple instances of the same software component must be loaded and isolated from one another.

The cost of user-defined class loaders is a consequence of the inability of current JVM implementations to share the main-memory representation of a class between multiple definitions of that class. JVMs typically construct the executable image of a program at runtime in several incremental steps: first by loading class files from locations specified by their loaders and building corresponding main memory representations, then by linking them to other classes as symbolic references are encountered during method execution, and eventually by compiling performance-critical methods. This evolution of a program image is a consequence of dynamic binding and of the use of an architecture-neutral format of class files, as required by the specification of the Java programming language [10] (JLS). Both make the building of a sharable image at runtime difficult and prevent applying well-established shared library techniques used for less dynamic programming languages (e.g. [3]).

Several recent efforts have achieved some degree of sharing of the runtime representation of classes between executing programs [5, 6, 17, 8]. However, they can only do so when dynamic binding has a predictable behavior across loaders, e.g., when a symbolic link from a class A to a class B is guaranteed to resolve identically across all class loaders. Thus, sharing is only supported for those class loaders whose behavior is fully controlled by the JVM. Sharing is not supported when classes are defined by user-defined loaders. Systems based on static compilation (e.g., [18]) face similar issues (see Section 6).

This paper presents a novel technique for sharing the runtime representation of classes between multiple arbitrary defining class loaders, and describes its application to the Multi-Tasking Virtual Machine (or MVM) [5]. This new implementation of MVM is called hereafter CLSVM (*Class Loader Sharing Virtual Machine*). CLSVM is a significant step forward in the technology of transparent sharing of safe language meta-data, and improves on MVM by bringing the benefits of sharing to user-defined class loaders.

CLSVM's sharing of the runtime representation of a class is orthogonal to sharing by delegation. Sharing by delegation makes a class type visible to mul-

[1]The character strings representing symbols are usually shared across representations of all classes though.

tiple loaders and has an impact on program semantics, while sharing of the runtime representation of classes is transparently and automatically performed by the JVM, has no impact on program behavior, and does not violate or impact type safety.

In CLSVM, sharing is achieved by splitting the runtime representation of a class into loader-dependent and loader-independent parts, and by making bytecode interpretation *loader re-entrant* so that the bytecodes of methods are sharable across multiple loaders. The loader-dependent part is replicated for each loader that defines the class. Classes loaded by the bootstrap loader are treated specially in order to exploit the predictability of symbolic link resolution and consequently to maximize sharing.

A consequence of such a design is that some of the class loading and run-time compilation effort is shared across multiple class loaders. In particular, class file parsing, and most of the building of a main-memory runtime representation of the class is done only once. Post-processing of bytecodes that may take place at link time is also done once.

CLSVM's dynamic compiler mixes two strategies to reduce the overhead of dynamic compilations. Whenever possible, the compiler attempts to share compiled code across multiple defining loaders. When sharing is not possible, the compiler constructs a new version of the compiled code. Even in this situation opportunities for compilation time savings arise: only if the method has not been compiled at all is it compiled from its bytecodes. Otherwise, instead of recompiling the method, the compiler clones the compiled code and modifies its loader-dependent part.

The rest of this paper is organized as follows. Section 2 describes principles for sharing between multiple loaders. Section 3 details a prototype implementation of CLSVM based on MVM, which in turn extends the Java HotSpot$^{TM}$ virtual machine (referred to as HSVM). Section 4 discusses how dynamic compilations take advantage of sharing across loaders. Section 5 reports a quantitative assessment of the impact of sharing on the performance and memory footprint of programs written in the Java programming language (or *Java programs*). Related work is discussed in Section 6. Section 7 summarizes the contributions of this work.

## 2 Design Overview

Sharing an element of the runtime representation of a class between class loaders is possible when the element is independent of its defining loaders. Loader dependencies arise from symbolic links to other classes, which may resolve at runtime to different class definitions from one loader to another, and from references from a class's runtime representation to data that are private to a loader, such as instances of `java.lang.Class` or static variables. At first sight, a significant amount of the runtime representation of a class appears independent of the loader that defines it. In particular, it should be possible to share across two loaders, each defining a class described by the same class file, the bytecodes of

that class, the constant part of its constant pool (i.e., the part that can never resolve to different objects at runtime, such as constant values), and meta-data describing the class itself (e.g., tables holding description of fields, methods, and exceptions).

However, several obstacles make sharing across multiple loaders difficult. First, loaders may resolve the super class of the same class differently. This may result in different object layouts (since one super-class may declare more fields than the other, and of different types), and different virtual tables (since one super-class may declare a different number of methods, with different signatures, and different methods may be overridden). Second, interpreters often exploit resolved symbolic links to rewrite bytecodes into faster versions that need not test whether links should be resolved or whether classes should be initialized. Such *quickened* bytecodes become, in effect, loader-dependent. The dynamic compiler also exploits information derived from resolved symbolic links and makes compiled code loader dependent. These obstacles taken together seem to contradict the first intuition that a significant amount of meta-data can be shared across loaders.

Our design overcomes these problems as follows. First, sharing is allowed only if some conditions on inheritance are met, so as to avoid dealing with cases where object layout and virtual tables would be different. Second, the interpretation of bytecodes is made loader re-entrant by re-organizing the runtime representation of classes so as to efficiently access each loader's private data, and by adding *barriers* to guarantee that link resolutions and class initializations are performed upon their first use by a loader. Third, the dynamic compiler maintains information to help determine if native code can be shared between loaders, and if not, to help build a new version of compiled code without paying the cost of a full-blown compilation from the method's bytecodes.

### 2.1 Terminology and Notation

We use the terminology and notation of [13] to describe relations between classes and class loaders. Namely, a class type is denoted as $< C, L_d >^{L_i}$, where $C$ denotes the name of the class, $L_d$ denotes the class's *defining* loader, and $L_i$ denotes its *initiating* loader. The initiating loader of a class (i.e., the loader invoked to load the class) is not necessarily the loader that defines the class, due to the API of class loaders that enables one loader to delegate the definition of a class to another. The simplified notation $< C, L_d >$ is used when the initiating loader of a class is not relevant. Similarly, $C^{L_i}$ denotes a situation when the defining loader is not relevant. By definition $< C, L_1 >=< C, L_2 >\Rightarrow L_1 = L_2$.

We extend this notation with the $\sim$ operator to denote that two class types satisfy the *same sharing conditions* (see Section 2.2) and can therefore share their runtime representation. By definition $< C, L_1 >\sim< C, L_2 >\Rightarrow< C, L_1 >\neq< C, L_2 >$ (this restriction simplifies discussion that follows). For convenience, we introduce the following notation: $< C, L_1 >\cong< C, L_2 >$ to denote $< C, L_1 >\sim< C, L_2 > \vee < C, L_1 >=< C, L_2 >$.

### 2.2 Sharing Conditions

Let us consider two distinct loaders $L_1$ and $L_2$, each defining a class $C$. The conditions that enable $< C, L_1 > \sim < C, L_2 >$, that is, the sharing of the runtime representations of $< C, L_1 >$ and $< C, L_2 >$, are defined below.

The first condition for $< C, L_1 > \sim < C, L_2 >$ is that the class files submitted to the JVM by $L_1$ and $L_2$ must be identical. For simplicity, two class files (either residing on a disk or dynamically generated) are considered identical if the classes are byte-per-byte equal, although this is a fairly coarse method for determining if two class files encode the same class definition. A more precise implementation would require parsing the class files and determining equivalence of the declarations. This approach is substantially more expensive and, in practice, is unlikely to be much more effective.

The second condition requires that the super-classes $S^{L_1}$, $S^{L_2}$ of $< C, L_1 >$ and $< C, L_2 >$ respectively are such that $S^{L_1} \cong S^{L_2}$.

The third condition is that $< C, L_1 >$ and $< C, L_2 >$ have the same abstract methods. Note that this property holds if the two sharing conditions above are satisfied and if $C$ does not implement any interfaces. The third condition is specifically directed at classes that implement interfaces. It guarantees that all classes that share their runtime representation have the same *unimplemented methods*, that is, methods that are not defined by the class but yet are declared in an interface. For example, let us consider a class $C$ that implements an interface $I$ that declares a single method $m$. Method $m$ is unimplemented if neither $C$ nor any of its super-classes implement $m$. In this case, $m$ must be treated as a public abstract method of $C$.

The sharing conditions guarantee several properties that simplify sharing. First, they guarantee that $< C, L_1 >$ and $< C, L_2 >$ have the same number of static variables, that their respective instances have the same number of fields (whether directly defined by the class or inherited), and that fields with the same name have the same signature and will be at the same offset. This property allows the JVM to lay out identically the instances of classes whose runtime representation is shared, to share reference maps used for garbage collections, and to share the descriptions of fields. Second, the sharing conditions guarantee that the methods of $< C, L_1 >$ and $< C, L_2 >$, whether declared directly by these classes or inherited, have the same name, signature, protection level, and bytecodes, and that they can be assigned the same index in a virtual method table (the Java programming language inheritance model enables table-driven implementation of virtual method dispatch).

Although the sharing conditions guarantee that $< C, L_1 >$ and $< C, L_2 >$ implement the same number of interfaces, and have the same unimplemented methods, they do not mandate that these interfaces declare exactly the same methods. Consider the example in Figure 1. In this case, the sharing conditions allow the runtime representation of $C$ to be shared across $L_1$ and $L_2$, although they implement different interfaces.

```
interface A { // In L1          interface A { // In L2
   int foo(int i);                Integer foo(Integer i);
   long foo(long l);              void bar(int i);
   void bar(int i);
}                               }
abstract class C implements A { // Defined by both L1 and L2
   int foo(int i){...}
   Integer foo(Integer i){...}
   long foo(long l){...}
} // bar is the only unimplemented method in both L1 and L2
```

**Fig. 1.** The sharing conditions can be satisfied despite different interface definitions.

# 3 The Design of CLSVM

CLSVM extends MVM, the multi-user, multi-tasking virtual machine [5, 7] with the ability to share meta-data across user-defined class loaders. The parts of the design of MVM relevant to CLSVM are briefly reviewed before discussing the actual design of CLSVM.

## 3.1 Background on MVM

MVM is an implementation of the JVM that co-locates the execution of multiple programs in a single operating system process. Each program execution is carried out by an *isolate* [11]. Isolates provide a program with the illusion of a standalone JVM: programs have the same behavior as if they were running on a JVM of their own. Each isolate has its own primordial loader and hierarchy of class loaders. No sharing of objects can take place between isolates and the JVM safeguards against most types of inter-isolate interference.

MVM substantially reduces the footprint of programs by implementing a form of sharing that we call *task re-entrance*. Task re-entrance is supported only for classes defined by class loaders whose behavior is fully controlled by MVM, that is, the *primordial* and *system* loader of each isolate.

The primordial loader is a special class loader that bootstraps the class loading mechanism. It is used to load the *base* classes that are intimately associated with a JVM implementation and are essential to its functioning (such as classes of the `java.*` packages). The system loader is the loader that defines the main class of a program. It typically obtains class files from the local file system at a fixed location specified at program start-up. MVM forces this location to be the same for all programs it executes, and requires that class files stored there remain unchanged for the duration of its execution. The system loader serves class loading requests by first delegating them to the primordial loader, and only defines classes that the primordial loader failed to define. This behavior is predictible, and a class loaded by a primordial or a system loader of any task is always built from the same class file. Further, symbolic references from classes

defined by a primordial or a system loader always resolve identically across all tasks.

This allows for a simplified form of sharing where only the mutable state part of the runtime representation of a class (e.g., static variables, class initialization state, protection domain, instance of `java.lang.Class` etc.) needs to be replicated per loader. In particular, information derived from resolved symbolic links, such as field offsets, virtual table indexes, static method addresses, etc., can be shared across loaders, further increasing the amount of sharing. In MVM no form of sharing is supported for classes defined by program-defined loaders.

Like MVM, CLSVM implements task re-entrance for classes loaded by primordial loaders. For all user-defined loaders CLSVM implements *loader re-entrance*, which allows sharing of the runtime representation of a class with any other class that satisfies the sharing conditions described earlier. The type of re-entrance implemented for system loaders can be chosen at start-up time: by default, CLSVM uses task re-entrance, like in MVM. All other aspects of MVM, such as isolate management and termination, per-isolate garbage collection, and fast inter-isolate communication, are left unchanged in CLSVM.

The rest of this section focuses on the following aspect of loader re-entrance: (i) how to organize the runtime representation of a class so as to maximize sharing while minimizing its overhead, (ii) how to efficiently retrieve loader-private information from shared code, and (iii) how to make the interpretation of shared bytecodes loader re-entrant.

### 3.2 Runtime Representation of Classes

The runtime representation of a class consists of data structures that mirror the architecture-neutral binary representation of that class, in a main memory format optimized for the various sub-systems of the JVM.

In CLSVM, the runtime representation of a class is split in a loader-independent and a loader-dependent representation (*LIR* and *LDR*, respectively). Loaders that satisfy the sharing conditions for a class share the same LIR, but each has its own LDR for the class. LIRs include a reference to a LDR *template*. The template serves two functions: it is a blueprint for constructing an LDR, and, to minimize space overhead, it is always used as the LDR of one loader.

Figure 2 illustrates this organization. It depicts the runtime representation of two classes satisfying the same sharing conditions: $< B, L_1 >$, whose LDR acts as a template, and $< B, L_2 >$ that was built using $< B, L_1 >$.

The LIR contains most of the runtime representation of a class. It consists of a `sharedRep` object which includes a reference map for garbage collection, references to an array of fields declared by the class template, to a shared constant pool, and to the LDR currently used as a template.

Each `sharedRep` of a LIR $S$ also includes a reference to the *super* `sharedRep` of the LIR shared by the runtime representation of the super-classes of all classes that have $S$ for their LIR. Recall that the second sharing condition requires that the super-classes of two classes $< C, L_1 >$ and $< C, L_2 >$ such that $< C, L_1 > \sim < C, L_2 >$ are either the same class or share their runtime

representation. In either case, this means that the `sharedRep`s of the superclasses of $< C, L_1 >$ and $< C, L_2 >$ are the same.

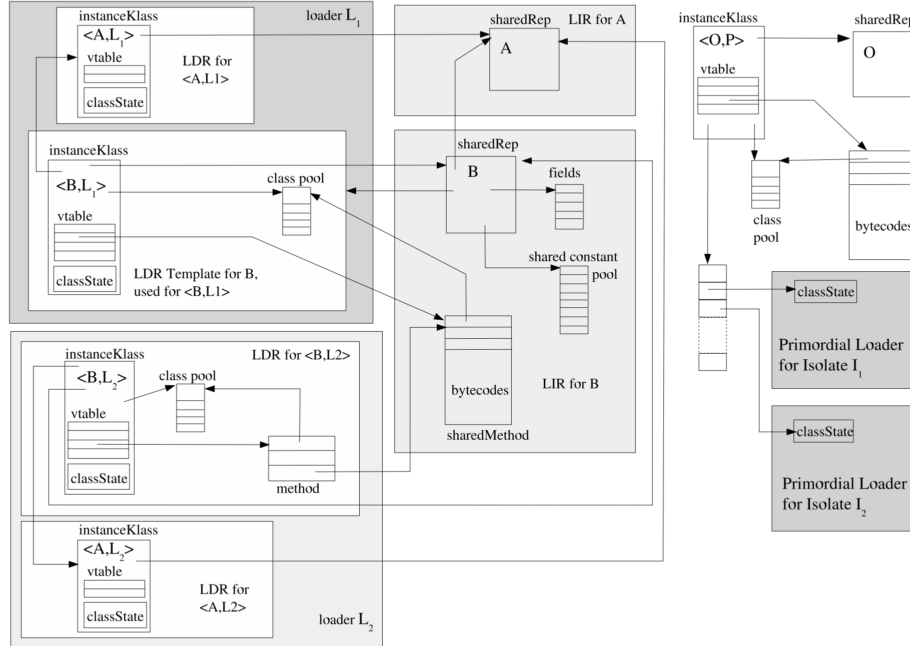


**Fig. 2.** CLSVM runtime representation of classes.

The LDR of a class consists of an `instanceKlass` object, which includes storage for its virtual method table (vtable) and its interface table (itable), and a reference to the `sharedRep` object it was built from. The runtime representation of classes that are task re-entrant is structured slightly differently in that most of the LDR is also shared across the task re-entrant loaders, except for non-re-entrant state, such as the class's static variables, the instance of `java.lang.Class` and `java.lang.ClassLoader` for the class, and its initialization state. This non-re-entrant state is embedded in the `instanceKlass` object for loader re-entrant classes. This arrangement is not possible for task re-entrant classes since the whole `instanceKlass` object is shared among all isolates. Instead, the non-re-entrant state of each isolate sharing the `instanceKlass` is stored in a separate object accessible via a table indexed by isolate identifiers; the reference to the table is stored in the `instanceKlass`.

An example of this is shown on Figure 2: a class O is defined by the primordial loaders of two isolates, $I_1$ and $I_2$. The resulting class type $< O, P >$ is represented by the same `instanceKlass` for both $I_1$ and $I_2$, and only task-dependent state is replicated for each task. Regardless of whether their class is task re-entrant or not, class instances (i.e., Java objects) contain in their header a pointer to the `instanceKlass` that represents their class.

As mentioned earlier, the sharing conditions guarantee that the vtable indexes are the same across all loaders that satisfy them. They also guarantee that methods are inherited and overridden in exactly the same way across class loaders. However, entries in vtables must refer to loader-dependent method representations. Methods have two runtime representations: a loader-dependent representation, called a method object, and a shared method object. Method objects consist of an invocation counter and references to a shared method object, to a class pool, and to native code produced by the dynamic compiler, if any (see Section 4). Shared method objects encapsulate the bulk of a method definition, most notably, the method bytecodes. All of the shared method object is loader-independent, except for a loader-specific header. This header comprises the same information as in method objects, except for the shared method object reference. This organization allows one `instanceKlass` (typically, the LDR template's) to use the shared methods directly (like$< B, L_1 >$ on Figure 2), thus avoiding the space overhead of systematically splitting the runtime representation of a method.

The interpreter and the compiler dynamically resolve links using the class pool, the shared constant pool, and the constant pool cache of the class that defines the method being executed. The class pool and the shared constant pool are built directly from the constant pool defined in a class file. The class pool is filled only with symbolic links to classes. All other constant pool entries are entered into the shared constant pool, which contains only loader-independent information, namely indexes to class pool or shared constant pool entries, constant numerical and string values, or symbols. Loader-dependent information, other than symbolic links to classes, is confined to the constant pool cache, constructed at class link time. It contains information built from resolved symbolic links to fields, methods, and interfaces to enable faster interpretation of some bytecodes.

### 3.3 Class loading

Classes loaded by the JVM are recorded in a *system dictionary* that maps keys composed of a fully qualified class name and a class loader reference to an `instanceKlass`. Multiple entries in the dictionary can refer to the same `instanceKlass` as a result of delegation between loaders.

A *shared class repository* maps linked lists of `sharedRep` objects to unique fingerprints computed over the bytes of class files. All `sharedRep`s in a given linked list are constructed from class files that all have the same fingerprint value. Having more than one shared runtime representation of a class for the same class file can occur because of possible violations of the sharing conditions between defining loaders (e.g., if the shared representation of the super classes of the defined classes are different). Fingerprints are computed as SHA-1 digests [16] of class files.

The class loading machinery uses the system dictionary and the shared class repository to determine whether a loader's request for defining a class can use an existing shared runtime representation for the newly defined class. When

instructed by a loader to define a class, CLSVM fetches a class file from the specified input stream and computes its SHA-1 digest. The digest is used to retrieve all the shared representations of classes that were built with a class file of equal value. Note that the format of the class file does not need verification before computing the SHA-1 digest, since if the specified class file does not conform to a valid class file format, its digest cannot map to an existing entry of the shared class repository. If the digest does not map to any `sharedRep`, the format of the class file is verified and parsed to create a new `sharedRep` object, which is then entered into the repository.

If a linked list of `sharedRep` objects is found in the repository, an element that satisfies the sharing conditions for the defining loader is looked up. Let $L$ be that loader, and $< C, L >$ the class type being defined. The first of the sharing conditions already holds since all `sharedRep`s from the list have a digest equal to that of the class file submitted by the loader. To test the second condition, the system dictionary is searched for the `instanceKlass` of $< C, L >$'s super class. If the search is not successful, class loading proceeds recursively to load the super-class. Once the `instanceKlass` of the super class is retrieved, its reference to its `sharedRep` is compared with the `super` reference of the `sharedRep` under evaluation. The second sharing condition holds if the two references are equal.

Lastly, $< C, L >$ must define the same unimplemented methods as other classes already sharing the evaluated `sharedRep`. Because the first two sharing conditions already hold, only unimplemented methods of $< C, L >$'s declared interfaces need to be verified. Hence the third condition is automatically satisfied if $< C, L >$ does not declare any interfaces. If it does, then unimplemented methods are looked up in the `sharedRep` and compared to those of $< C, L >$. The third condition holds if the number of unimplemented methods and their names and types are the same.

If none of the existing `sharedRep`s satisfies the sharing condition for $< C, L >$, a new one must be created. Creating a new `sharedRep` in this case does not require parsing the class file. Instead, an existing `sharedRep` is cloned and changed only in those places that depend on the super class and the unimplemented abstract methods, since a violation of the sharing conditions corresponds to having different values for some of these.

### 3.4 Sharing bytecodes

The runtime representation of classes described above is not sufficient to allow sharing of methods. Bytecode interpretation must also be made loader reentrant. This requires efficient access to a class loader's copy of static variables, and proper triggering of link resolution and class initialization once for each loader that shares the bytecodes. Both are achieved by using the constant pool cache associated with the private representation of the *current class*, i.e., the class that defines the method being executed. The reference to the current class is stored, upon method invocation, in a dedicated location on the invoked method's stack frame. Short sequences of instructions called barriers trigger link resolution and class initialization.

A link resolution barrier (LRB) is required for all bytecodes that refer to a loader-dependent symbolic link. In CLSVM, the bytecodes in this category are the quickened versions of `getfield`, `putfield`, `invokevirtual`, `invokespecial`, `invokeinterface`. LRBs are redundant in presence of class initialization barriers, so the former is not necessary if the latter is already required. A class initialization barrier (CIB) is needed when interpreting the quickened versions of the four bytecode instructions that might trigger class initialization: `getstatic`, `putstatic`, `invokestatic` and `new`.

Both LRBs and CIBs work along the same principle: the operand of a quickened bytecode is an index to an entry of the current class's constant pool cache that holds information necessary to interpret the bytecode (e.g., a field offset, a vtable index, etc.). The information is initialized with a distinguishable marker that is tested by the interpreter. On SPARC® processors, these barrier tests add only a single branch on register value. For instance, offsets and vtable index information are typically initialized to a negative value so that LRBs just consist of a single branch on negative value, as shown below:

```
ld        [Rcache + // Retrieve offset to field
            (header_size + 2*wordSize)], Roffset
brgz,pt   Roffset, resolved // LRB
ld        [ Robject + Roffset], Rvalue // load field
```

Upon detecting the marker by a barrier, execution is routed to a stub that calls the runtime to perform the action associated with the barrier (link resolution or class initialization). Before resuming interpreted execution, the marker is replaced with the information needed by the interpreter in the constant pool entry, so that subsequent interpretation of bytecodes indexing that entry with the same class loader will not trigger link resolution. The first interpretation of the same bytecode instruction, but on behalf of a different loader, will trigger the barrier again since each loader uses a distinct constant pool cache.

## 4 Impact on Dynamic Compilation

HSVM mixes bytecode interpretation with dynamic compilation to achieve high-performance. Methods are initially interpreted. Per-method invocation counters are incremented on each invocation to detect frequently called methods. Methods that reach a given threshold of interpreted invocations are compiled. Subsequent invocations result in executing their compiled code.

This approach can be improved so that the effort of compiling the methods of a class is amortized across the loaders that define it. One possible strategy is to make the code produced by the dynamic compiler loader re-entrant, so that different defining loaders of the same method *always* share the same native code. This strategy has several advantages. First, compilation costs are paid only once per shared representation of a method, no matter how many loaders define that method, hence the cost of compilation is amortized across loaders. Second, because the code is re-entrant, it can be used immediately by any loader defining

the method, eliminating bytecode interpretation. Third, memory footprint is reduced by sharing the compiled code across loaders.

However, making compiled code loader re-entrant introduces some overhead otherwise eliminated by a dynamic compiler. Dynamic compilation exploits the runtime knowledge of resolved links to remove the overhead of dynamic linking. For example, a dynamic compiler can determine the offset of a field of an object and generate a simple load instruction that does not use any meta-information (such as the runtime constant pool cache) at runtime. Such optimizations are not possible with loader re-entrant code because a level of indirection is required wherever a symbolic link to another class is used. So, for instance, loading a field of an object requires determining, at runtime, the current loader and then finding out the offset to the field in the context of that loader. Whereas the impact of such indirection may be benign to the performance of interpreted methods, it may be prohibitive in compiled code.

CLSVM employs a three-pronged strategy for dynamic compilation that mixes task re-entrance, cloning, and sharing of loader-dependent code.

As in MVM, methods of task re-entrant classes are compiled into task re-entrant code. Such code is produced by adding class initialization barriers before every possible first use of a class, and generating code to access static variables in a task re-entrant way. That is, static variables are retrieved from the class's table of per-isolate class state using an isolate identifier stored in a current thread's descriptor (see [5] for details). Past experience with MVM showed that the impact of task re-entrance on performance is negligible compared with the benefits of sharing compiled code for method of classes defined by task re-entrant loaders.

For methods of loader re-entrant classes, the compiler produces code optimized for a particular set of loader dependencies, i.e., using information derived from symbolic links as resolved by a particular loader. The code is annotated along the way with information identifying the loader-dependent sequences of instructions and what they depend on. The annotations are then used to determine if the code can be used as is by other loaders, or if a new version should be produced according to a new set of dependencies. In the latter case, instead of compiling the method for each defining loader from scratch, the compiler clones an existing version of it, and modifies its loader-dependent part only. This makes generation of native code faster as steps for parsing bytecodes, building an intermediate representation, performing optimization, and generating code are avoided.

Loader-dependent code sharing is based on the observation that the loader-dependent information derived from symbolic links may be constant across loaders that share the runtime representation of the class that defines the compiled method. For example, the offset to an instance variable of a class $B$ is constant across all defining loaders of $B$ that satisfy the same sharing conditions. In other words, a symbolic link to an instance variable of $B$ from a class $A$ is constant across two loaders $L_1$ and $L_2$ that share the runtime representation of $A$ (i.e., $< A, L_1 > \sim < A, L_2 >$) if $B^{L_1} \cong B^{L_2}$. Thus, if the only symbolic links used in a

method $m$ of $A$ are to instance variables of B, the compiled code for $m$ can be shared between $< A, L_1 >$ and $< A, L_2 >$. In this strategy a method is compiled from bytecodes only once, no matter how many loaders define its class. Once compiled, native code for the method is obtained through cloning or sharing.

The compiler assumes the linkages of the loader that triggered the compilation, and exploits information obtained from resolved links, effectively making the native code dependent on a (potentially empty) set of resolved links. To enable sharing and cloning, the location of each loader-dependent sequence of native instructions is recorded along with the position of the corresponding bytecode instruction. This position offers a compact and loader-independent way of documenting a dependency: given a bytecode position and a loader-dependent method representation, one can retrieve the type of its dependency (e.g., offset of an instance field), and the class the dependency refers to (through the class pool of the method's class). Thus, code annotations consist of pairs of offsets: an offset to the first instruction of a loader-dependent sequence of native code, and another offset to a bytecode instruction. These pairs are recorded in a table of dependencies.

The table of dependencies is used to determine if native code listed in the shared method, and produced for a particular loader, can be used by another loader that shares the runtime representation of the method. Sharing of native code is possible if there are no dependencies, or if loaders have exactly the same dependencies. For example, a method that only manipulates instance variables of its class is loader-independent since the sharing conditions guarantee that offsets to these fields are the same across loaders that satisfy the same sharing conditions. In another example, a method may refer only to symbols of classes defined by the primordial loader in all loaders. In this case, the native code for the method can be shared between loaders since symbolic links to methods and variables of task re-entrant classes will refer to exactly the same item in all loaders.

The sharing of compiled code can be permitted even if the classes referred to by the code are not the same. For example, let us consider classes $A$ and $B$:

```
class A {
  private int x;
  private static int X;
  int getx(){ return x;}
  int getxX(){ return x * X;}
  int foo(C c){ return x * c.z;}
  int bar(){ return x * B.Y;}
}
```

```
class B {
  static int Y = 94;
}
```

Let us further assume that class $A$ has been defined by two loaders $L_1$ and $L_2$ such that $< A, L_1 > \sim < A, L_2 >$. Although $< A, L_1 > \neq < A, L_2 >$, the native code produced for method `getx` can be shared between $< A, L_1 >$ and $< A, L_2 >$ since the only dependency of the compiled code is the offset to the instance variable `x`, which is guaranteed by the sharing conditions (see Section 2.2) to

be the same for both loaders. Such symbolic link references do not need to be recorded in the dependency table. By contrast, the native code produced for the `getxX` method cannot be shared between the loaders as it depends on the address of the static variable `X` which differs for each defining loader. The case for `foo` is more subtle: if $C^{L_1} \cong C^{L_2}$, then the native code can be shared since `z` resolves to the same offset for both loaders, either because $C^{L_1} = C^{L_2}$, or because the sharing conditions guarantee this. Otherwise, the method cannot be shared. Similarly, the native code for the `bar` method can be shared between $L_1$ and $L_2$ if $B^{L_1} = B^{L_2}$. More generally, let $L_r$ be a loader that requests native code for a method $m$ of class $A$, $L_u$ the loader of one of the users of native code of method $m$, and $C$ a class on which $m$ depends. Table 1 lists, for each dependency type, the conditions for leaving the corresponding instructions unchanged, and what changes are required otherwise. If no condition is violated, no change is needed, and the code can be shared by $L_u$ and $L_r$.

To determine whether any changes to the code are required, the compiler iterates over the dependent class listed in the dependency table. For each dependent class $C$, the compiler first determines if the link to $C^{L_r}$ has been resolved, by examining the entry in $L_r$'s class pool at the index recorded in the dependency table. If the link to $C^{L_r}$ is not resolved, the determination for code sharing cannot be made, and sharing is prohibited. The compiler then selects the condition that determines if code change is needed based on the dependencies for class $C$ (see also Table 1). If the condition $C^{L_r} = C^{L_u}$ is required, the compiler compares the references to $C^{L_r}$ and $C^{L_u}$ obtained from the class pool of $< A, L_r >$ and $< A, L_u >$ respectively, at the index recorded in the dependency table. If the condition $C^{L_r} \sim C^{L_u}$ is required, the compiler compares the references to the `sharedRep` objects from $C^{L_r}$ and $C^{L_u}$.

**Table 1.** Modifications required for each type of dependencies to adapt a clone of a compiled method to a new loader $L_r$ is the requesting loader, $L_u$ is an owner of the original compiled code.

| Type of symbolic link | Conditions for leaving code unchanged | What to change if condition is false |
|---|---|---|
| instance variable | $C^{L_r} \cong C^{L_u}$ | offset in load/store instruction |
| dynamically bound method | $C^{L_r} \cong C^{L_u}$ | reset inline cache for virtual method/interface invocation |
| static variable | $C^{L_r} = C^{L_u}$ | address of static variable |
| class | $C^{L_r} = C^{L_u}$ | class address and instance size in immediate value register load |
| statically bound method | $C^{L_r} = C^{L_u}$ | address of method entry point in call instruction |

To clone the native code of a method, the compiler first copies the native code and walks over the dependency table to identify classes for which changes are required in the native code. For each such class, the compiler iterates over

the corresponding dependencies section. Using the bytecode position recorded there it retrieves the corresponding bytecode from the method template. Each such bytecode maps to a function that implements the logic for modifying the sequence of native code according to the operands of the bytecode instruction and the class pool of the recipient of the clone.

Since native code cloning or sharing is cheaper than compilation from bytecodes, switching from bytecode interpretation to native code execution takes place earlier for methods that have been already compiled.

## 5 Experiments

This section compares MVM and CLSVM with respect to memory footprint, program start-up time, and application execution time.

The aim of the presented techniques is to reduce the footprint of programs that extensively rely on user-defined class loaders, by sharing the runtime representation of classes across loaders of the same or of different programs. Another equally important goal is to avoid performance and start-up time regression with respect to MVM. Our prototype of CLSVM implements task re-entrance for both the primordial and system loaders, and loader re-entrance for user-defined loaders.

CLSVM is a derivative of MVM, which in turn derives from HSVM (the Java HotSpot$^{TM}$ virtual machine) version 1.3.1 with the client compiler. All results are reported relative to HSVM. The experiments were performed on a Sun Blade 1000$^{TM}$ equipped with 8 GB of main memory and two UltraSPARC® III+ processors clocked at 1015 MHz, running the Solaris$^{TM}$ 10 Operating Environment.

### 5.1 Start-up Time

In both MVM and CLSVM programs can be started either from the command line or directly (programmatically) by an isolate. The former option involves the creation of a process that communicates with a *login* isolate to request the launching of a new isolate and to establish input/output bindings between the isolate and the initiating process (see [7] for further details).

Start-up time can be approximated by running an empty program (i.e., one whose `main()` methods consists of just a `return` statement). Table 2 reports the results for CLSVM and MVM, for both ways of starting a program (*cli* for command line start-up, and *java* for start-up from within an isolate). The results are expressed as speed-up with respect to the time to execute the empty program with HSVM (e.g., in *cli*, MVM starts a program 1.84 times faster than HSVM). They indicate that support for loader re-entrance has no negative impact on start-up performance. This is expected since none of the features of loader re-entrance are exercized at start-up.

**Table 2.** Start-up improvements relative to HSVM.

| | cli | java |
|---|---|---|
| MVM | 1.84 | 26.82 |
| CLSVM | 1.85 | 26.21 |

### 5.2 Footprint

To quantify the impact of the design of CLSVM on memory usage, we experimented with two popular real-world applications: Apache Ant (version 1.6.2), and Apache Tomcat servlet engine (version 4.2.1). The latter was used to run JSPWiki [2], a Wiki clone implemented with Java Server Pages [1]. Tomcat maintains a hierarchy of class loaders to allow its components and applications it hosts to access different repositories of available classes and resources. Ant uses class loaders in a similar fashion, to customize per-user access to different libraries and resources.

Another common use of class loaders is to transparently inject code at runtime, either for profiling purposes or to enhance application code with an aspect (e.g., persistence). Typically a loader uses a bytecode editing library to transform the contents of fetched class files and submits the modified bytecodes to define the class. Our third experiment, referred to as the *bytecode transformation workload*, emulates this behavior using Apache's BCEL toolkit. We applied it to programs from the SPECjvm98 suite. The transformation was relatively simple: counting the number of dynamic (run-time) accesses to static variables by application code.

Memory footprint measurements were obtained with the help of the `pmap` command from the Solaris Operating Environment and then correlated with the JVM-specific runtime information regarding its use of virtual memory regions. Memory accounting was thus accurate for all memory regions, such as the heap area used for application data, runtime representations of classes, and compiled code area. The numbers reported exclude memory regions shared across processes, such as read-only parts of shared libraries and read-only memory-mapped jar files.

The data in the following figures were obtained as follows. For non-server tests (Ant, SPECjvm98 benchmarks), multiple instances of a given program were executed in sequence. Each program instance was artificially kept alive through a shutdown hook. Each hook would send a notification to an external supervising process then wait for an answer before exiting. Upon receiving a shutdown hook notification, the supervising process would obtain memory usage before starting the next program instance. Commands to exit were sent to the shutdown hooks only once all programs had been executed and their memory usage captured.

Recourse to shutdown hooks to maintain Wiki servers alive is unnecessary since these stay up and running until explicitly instructed to terminate. Therefore, the experiment proceeded by starting one Wiki server, submitting 100 requests, and capturing memory usage before repeating this sequence up to the

desired number of servers. The same mix of 100 requests was sent to each server. It consisted of requests for pages' content, for editing them, and either saving or cancelling the edits.

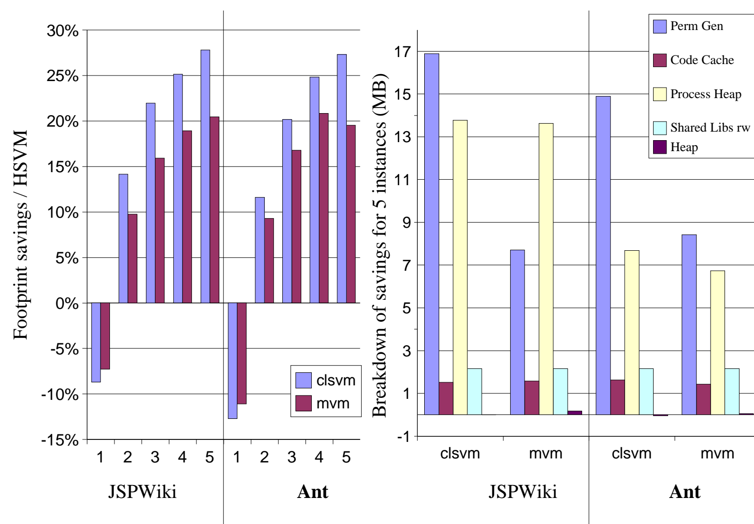


**Fig. 3.** Footprint savings for JSPWiki on Tomcat and for Ant. The left-hand part shows savings relatively to HSVM as the number of program instance grows. The right-hand part shows the breakdown of savings for 5 running programs.

Figure 3 shows the data obtained for Ant (up to 5 instances) and Tomcat/JSPWiki (up to 5 servers). The left-hand part of the figure shows memory footprint reduction relative to HSVM; the right-hand part shows a breakdown of the savings for 5 instances of the measured programs.

When executing a single program, both MVM and CLSVM fare worse than HSVM, because the memory overhead of the login isolate is not amortized. This results in footprint increase ranging from 7.2% to 12.7%. The overhead for CLSVM is larger than for MVM, because of (i) the additional data structures required for loader re-entrance (e.g., the shared class repository, SHA digests), which are not amortized due to the absence of sharing; (ii), the extra overhead due to the classes used by CLSVM to compute SHA digests of class files (e.g., `java.security.MessageDigest`); and, (iii), the systematic decomposition of the runtime representation of *all* classes, including task re-entrant ones, into loader-dependent and loaded-independent parts, which, in the absence of loader re-entrant sharing, brings no benefits.

As soon as more than one application is executed, the benefits of sharing outweigh the overhead of the login isolate, and both MVM and CLSVM reduce

the aggregate footprint of programs when compared to HSVM. The savings increase with the number of running programs, and increase faster with CLSVM than with MVM. The breakdown of savings shown in the right-hand part of Figure 3 offers insight into the reasons for this behavior. There are four sources of savings: the permanent generation (a garbage collected area that holds most of the runtime representation of classes); the code cache (where the compiled code produced by the dynamic compiler resides); the JVM's process C-heap (which stores some of the dynamic data structures of the JVM); and the private segments of the shared libraries used by the JVM.

The permanent generation and the process's heap are the main contributors to memory footprint reduction. Code cache can also contribute to savings, depending on the amount of re-entrant compiled methods. Sharing the runtime representation of classes across user-defined loaders mostly affects the permanent generation. For both Ant and Tomcat/JSPWiki, the permanent generation of CLSVM grows three times slower than that of MVM with each additional program instance. All other memory areas grow at almost the same pace.

The above demonstrates that CLSVM can bring benefits to these applications over and beyond what MVM already provides. CLSVM's gains, relative to MVM's, depend on the ratio of the number of classes defined by program-defined loaders (and their size) to the number of other classes. Both Ant and Tomcat/JSPWiki heavily exploit user-defined loaders: over 50% of classes are defined by them (see Table 3).

**Table 3.** Population of classes for the programs used in the experiments.

| defining loader(s) | Ant | Tomcat | *db* | *javac* | *mpegaudio* | *jack* |
|---|---|---|---|---|---|---|
| Primordial | 343 | 476 | 316 | 316 | 315 | 316 |
| System | 149 | 14 | 256 | 269 | 263 | 260 |
| User-defined | 540 | 806 | 9 | 149 | 57 | 60 |

The bytecode transformation workload gives a more contrasted picture due to the much smaller proportion of classes defined by user-defined loaders[2]: across the SPECjvm98 programs, the population of user-defined classes (i.e., the classes subjected to runtime bytecode transformation) is between 4 (*javac*) to 60 (*db*) times smaller than the population of other classes. The bytecode transformation tool alone accounts for between 35% to 42% of the total population of loaded classes. Despite this the sharing across loaders of CLSVM brings visible benefits.

The left-hand part of Figure 4 compares the footprint savings from MVM and CLSVM for a sample of the SPECjvm98 programs. The sample is chosen for its differing footprint characteristics: *javac* (respectively, *db*) has the largest (respectively, smallest) population of classes defined by user-defined loaders; *meg-*

[2]The bytecode editing loader rewrites only the classes from the SPECjvm98 programs, not the JDK classes these programs uses.

*paudio* and *jack* have almost the same population of loaded classes, but they vastly differ in terms of memory usage, as shown on the right-hand part of Figure 4: when run with HSVM, the heap in *jack* accounts only for 30% of the total footprint, compared to 50% for *mpegaudio*, and over 60% for both *javac* and *db*. As a result, the benefits of sharing are more pronounced for *jack* than for the other programs. Like before, the overhead of the login isolate is felt for the first program invocation, and erased as soon as more than one program is run. The footprint of *db* is initially worse with CLSVM than MVM because of the very small number of loader re-entrant classes. In this case, the overhead of the systematic decomposition of the runtime representation of classes into loader-dependent and loaded-independent parts slightly disadvantages CLSVM.

Likewise, when user-defined loaders are not used by programs, this decomposition adds an almost unnoticeable regression in footprint compared to MVM (less than 0.3% across all the SPECjvm98 programs).

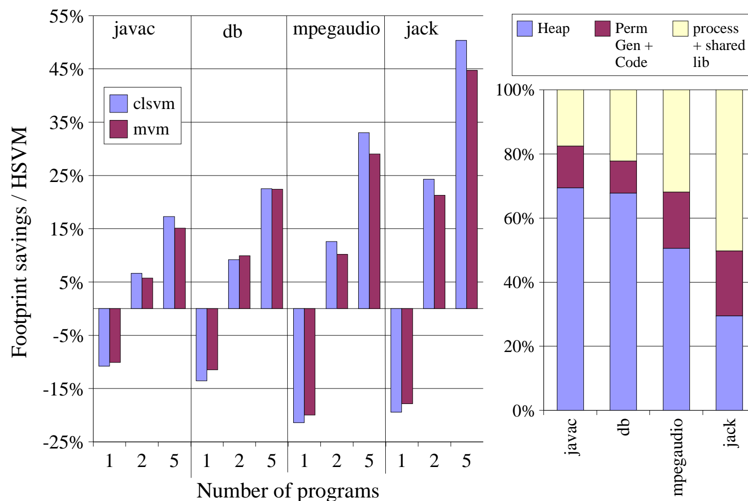


**Fig. 4.** Footprint savings for the bytecode transformation workload.

### 5.3 Performance

One of our goals is to ensure that loader re-entrance does not negatively impact performance when user-defined class loaders are not used, or when there are no opportunities for sharing. The left-hand part of Figure 5 shows the performance improvement relative to HSVM on a sample of the SPECjvm98 benchmarks.

Both CLSVM and MVM improve performance noticeably when compared to HSVM. This is due to two factors. First, methods of classes defined by the primordial and system loaders are compiled into task re-entrant code that is shared across multiple programs, whether these execute concurrently or serially. Thus, programs benefit from the elimination of dynamic compilation and interpretation costs. Second, the invocation counters of methods are shared across program execution, which allows identifying and compiling more hot methods that what can be identified with a single program execution. As a result, a larger set of compiled methods is available for programs in MVM and CLSVM than in HSVM.

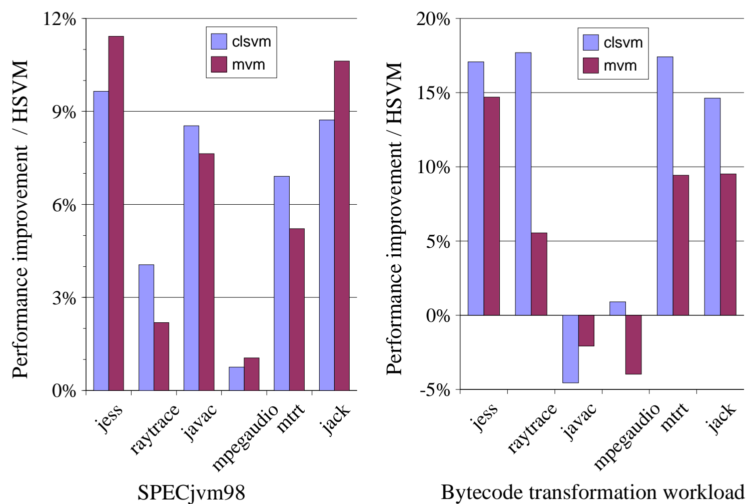


**Fig. 5.** Performance with respect to HSVM. The right-hand part shows performance improvement with programs from the SPECjvm98 programs. The left-hand part show performance improvement with the bytecode transformation workload.

CLSVM slightly degrades performance compared to MVM on these benchmarks. We attribute this to the overhead of dealing with two method representations, which adds runtime tests to the calling convention of interpreted methods in CLSVM.

CLSVM further attempts to share compiled code across user-defined loaders or to produce new code by cloning existing code and changing its loader-dependent part. The effect of this strategy can be observed on the right-hand part of Figure 5, which shows the performance of the bytecode transformation workload applied to a sample of the SPECjvm98 suite. CLSVM's strategy for amortizing the cost of dynamic compilation across user-defined class loaders im-

proves performance relatively to MVM. The impact depends on the proportion of loader-dependent compiled methods to the total number of compiled methods, and to the amount of loader-dependent code that can be shared. For example, 24% of compiled methods pertains to classes defined by user-defined loaders in *jess* and 30% for *raytrace*. However, only 12% of these are shared across loaders for *jess*, whereas nearly 45% are shared for *raytrace*. Thus *raytrace* benefits much more from CLSVM. The current prototype of CLSVM is not capable of sharing or cloning code that inlines loader re-entrant methods. In this case, the method needs to be recompiled. This limitation impacts directly *jess*, *javac*, and *mpegaudio*, since these have a substantial amount[3] of compiled loader re-entrant methods that inlines other loader re-entrant methods.

CLSVM outperforms MVM on *mpegaudio* because of differences in how static fields are accessed in both.

The frequency of *mpegaudio*'s accesses to static fields is about two orders of magnitute higher than for other SPECjvm98 benchmark. The difference in how static fields are accessed in both virtual machines explains why CLSVM outperforms MVM on this benchmark, and also why MVM performs worse than HSVM. Specifically, MVM incurs the cost of an additional level of indirection when accessing static fields, regardless of whether the class defining the variable is task re-entrant or not.

Performance data obtained with Ant (not shown here) indicate 3.3% improvement of CLSVM over MVM, bringing performance relative to HSVM from 26.4% to 28.8%.

## 6 Related Work

Our work relates to recent efforts to share the main-memory runtime representation of classes between programs. This form of sharing can reduce both the footprint of Java programs and factor out the runtime costs of transforming class file to an optimized, architecture-specific class type representation.

One approach to sharing consists of launching for each Java program a separate OS process to execute an instance of the JVM, and to store the sharable part of the runtime representation of classes in an area of memory shared by the JVM processes [6, 8]. What is made sharable varies according to implementations: at the minimum, the bytecodes of methods, which are by far the largest part of the runtime representation of classes are shared[4]. The immutable part of the runtime representation of classes, such as symbol constants and some of the meta-description of methods and fields, can also be shared [6]. Additional information necessary to reconstruct the mutable part of the runtime representation of a class can also be stored in the shared area, to avoid parsing the original class file when constructing the process-private part of the class's representation.

[3] 31%, 14% and 11% for, respectively, *jess*, *javac* and *mpegaudio*.

[4] The Java HotSpot Virtual machine 1.5.0 implements sharing of method bytecodes this way.

ShMVM-C [6] goes one step further by also making the output of the dynamic compiler program re-entrant so that multiple JVM processes can share it.

An alternative to storing the sharable part of a class in shared memory is to encode the whole runtime representation of classes in a binary format natively supported by the host OS's shared libraries mechanism. For instance, SLVM [17] encodes the main-memory representation of classes in the ELF format. The resulting binaries are relocatable. Loading and relocation are performed by the linker. Better memory utilization can result from this approach, due to the systematic copy-on-write policy implemented by the linker on any page of mutable sections of the shared library.

Another approach to transforming Java classes into shared libraries is exemplified by GCJ [9], a portable, optimizing, ahead-of-time compiler for the Java programming language. The run-time system of GCJ supports program-defined class loaders and dynamic class loading, but code loaded this way can only be interpreted. [18] describes an extension to GCJ that supports sharing of code compiled ahead of time across program even if the class is not linked to classes with the same definition in each program. To this end [18] re-introduces runtime functions and data structures commonly found in standard virtual machine (e.g., vtables are constructed at runtime and each class is associated with a table of links to external symbols that get filled up at class load-time).

MVM [5] tackles the memory footprint problem by collocating multiple Java programs in the same OS process, and executing them within a single JVM capable of multi-tasking. Most of the runtime representation of classes, including compiled code, is made program re-entrant and shared across all tasks. Interference among programs is prevented by replicating the program-dependent part of the runtime representation of classes.

None of the systems mentioned above is capable of sharing, either within the same program or across programs, the runtime representation of classes defined by arbitrary user-defined class loaders, especially when they edit bytecodes or generates class file at runtime.

Like MVM and CLSVM, Microsoft's .NET allows for isolated execution of multiple applications in the same process [14]. The platform can be configured to transparently share some meta-data, although not as aggressively as CLSVM. Decisions concerning the trade-off between memory footprint and performance can be made at deployment time. *Domain-neutral* deployments result in slower execution as non-static meta-data is shared while static data and static code are replicated. The additional logic that directs callers to the appropriate static code or data is thus needed to provide application isolation. The standard form of application deployment in .NET does not have this feature, thus potentially providing better performance at the expense of memory footprint.

QuickSilver [15] amortizes the cost of producing a high-performance runtime image of a program by compiling the methods of a class into a relocatable format and storing the result of the compilation in files. Compiled method files can be generated off-line, or when an instance of the JVM exits. Subsequent executions of the program load the compiled method files, if available, upon

the class definition. Before its use, the compiled code is subjected to validation tests that determine if the code can be re-used by the running program. If validation succeeds, the code is *"stitched"* according to the state of the running JVM. Otherwise, the code undergoes standard dynamic compilation. In its latest version [12], QuickSilver generates code that uses an indirection mechanism in order to make most of the compiled code of methods read-only, thus reducing the aggregate footprint of multiple JVM instances. However, the indirection mechanism makes the code dependent on the address of an indirection table specific to one class loader. As a result, code cannot be shared across multiple class loaders within the same JVM instance. This limits the usefulness of this approach, especially for server environments where class loader-based containers are often used.

## 7 Conclusions

Defining class loading policies is a commonly used feature of the Java programming language. The reliance on class loaders is likely to grow, partly due to a growing popularity of load-time bytecode transformations applicable to aspect-oriented programming. However, existing implementations of the JVM poorly support class loaders with regard to resource utilization.

This paper describes CLSVM, a multi-tasking implementation of the JVM capable of transparently sharing the runtime representation of classes, including their bytecodes and compiled code, across multiple defining loaders. Sharing is achieved by separating out the part of the runtime representation of a class that depends on symbolic link resolution and by making bytecode interpretation loader re-entrant. Re-entrance is implemented by the addition of class resolution and initialization barriers and by efficient access to loader-dependent components of the runtime representation of classes. The dynamic compiler exploits sharing by maintaining loader dependencies and using them to determine when sharing of compiled methods across loaders is possible. If the required sharing conditions are not met, the code is cloned to avoid its compilation from bytecodes.

The presented techniques enhance MVM so as to extend the scope and benefits of code sharing to classes defined by arbitrary class loaders, regardless of whether these loaders pertain to different programs or the same one. Further, classes subjected to bytecode transformation at runtime also benefit from sharing.

The impact of the mechanisms implemented by CLSVM on end-to-end application performance is highly dependent on the proportion of compiled loader re-entrant methods that can be shared. Performance relative to MVM varies between $-3\%$ to $+12.8\%$, the highest improvements corresponding to cases when sharing of loader-reentrant compiled methods can be exploited. When compared to the Java HotSpot virtual machine, the gains are between 0.9% to 17%. Application start-up time is almost identical for both CLSVM and MVM, and between 1.85 (command-line launching of applications) and 26 (programmatic launching of applications) times faster than for the Java HotSpot virtual machine.

Along with not degrading start-up time and bringing about a modest improvement in end-to-end performance compared to MVM, the main motivation for this work, was to decrease memory footprint of applications that exploit user-defined class loading. This goal has been achieved: for example, for applications like Ant and Tomcat CLSVM improves the memory savings already achieved by MVM by between 15% to 40%, bringing total memory footprint down by 11.6% to 28% with respect to the Java HotSpot virtual machine. When class loading mechanisms are not used, memory overhead relative to MVM remains below 3%.

## References


1. http://java.sun.com/products/jsp.
2. http://www.jspwiki.org.
3. J. Arnold. Shared Libraries on UNIX System V. In *Summer USENIX Conference*, Atlanta, GA, 1986.
4. D. Balfanz and L. Gong. Experience with Secure Multi- Processing in Java. Technical Report 560-97, Department of Computer Science, Princeton University, Sept. 1997.
5. G. Czajkowski and L. Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *ACM OOPSLA'01*, Tampa, FL, Oct. 2001.
6. G. Czajkowski, L. Daynès, and N. Nystrom. Code Sharing among Virtual Machines. In *ECOOP'02*, Malaga, Spain, June 2002.
7. G. Czajkowski, L. Daynès, and B. Titzer. A multi-user virtual machine. In *USENIX*, San Antonio, TX, 2003.
8. D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. S. John. Building a Java$^{TM}$ Virtual Machine for Server Applications: The JVM on OS/390. *IBM Systems Journal*, 39(1), 2000.
9. Free Software Foundation (FSF). GCJ: The GNU Compiler for Java., 2003.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java$^{TM}$ Language Specification*. The Java$^{TM}$ Series. Addison Wesley, second edition edition, Sept. 2000.
11. Java Community Process. JSR 121: Application Isolation API., 2003.
12. P. G. Joisha, S. P. Midkiff, M. J. Serrano, and M. Gupta. A framework for efficient reuse of binary code in java. In *International Conference on Supercomputing*, pages 440–453, 2001.
13. S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *ACM OOPSLA'98*, Oct. 1998.
14. Microsoft Corp. *Programming with Application Domains and Assemblies*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconprogrammingwithapplicationdomainsassemblies.asp, 2005.
15. M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a Quasi-Static Compiler for Java. In *ACM OOPSLA'00*, Oct. 2000.
16. US Department of Commerce. Secure hash standard, Apr. 1995.
17. B. Wong, G. Czajkowski, and L. Daynès. Dynamically loaded classes as shared libraries. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, 2003.
18. D. Yu, Z. Shao, and V. Trifonov. Supporting binary compatibility with static compilation. In *2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, pages 165–180, 2002.