Dynamically Loaded Classes as Shared Libraries: an Approach to Improving Virtual Machine Scalability

Bernard Wong
University of Waterloo
200 University Avenue W.
Waterloo, Ontario N2L 3G1, Canada
kw3wong@engmail.uwaterloo.ca

Grzegorz Czajkowski, Laurent Daynès
Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043, USA
{grzegorz.czajkowski, laurent.daynes}@sun.com

Abstract

Sharing selected data structures among virtual machines of a safe language can improve resource utilization of each participating run-time system. The challenge is to determine what to share and how to share it in order to decrease start-up time and lower memory footprint without compromising the robustness of the system. Furthermore, the engineering effort required to implement the system must not be prohibitive. This paper demonstrates an approach that addresses these concerns in the context of the Java™ virtual machine. Our system transforms packages into shared libraries containing classes in a format matching the internal representation used within the virtual machine. We maximize the number of elements in the read-only section to take advantage of cross-process text segment sharing. Non-shareable data are automatically replicated when written to due to the operating system's streamlined support for copy-on-write. Relying on the existing shared libraries manipulation infrastructure significantly reduces the engineering effort.

1. Introduction

The Java™ programming language [1] has, in less than a decade, become very popular for use in the development of a variety of personal and enterprise-level applications. Its features, ease of use, and "write once run anywhere" methodology make the lives of programmers easier and can considerably reduce program development time. These characteristics are often overshadowed by the Java platform's performance, as some users feel that its start-up time is too slow, its memory footprint is too large, and that in general the underlying resources are not used optimally.

Different types of users are affected in varying degrees by these problems. For the majority of personal computer users, footprint is not necessarily a critical issue as most machines sold today have sufficient amount of memory to hold a few instances of the Java virtual machine (JVM™) [2]. However, on servers hosting many applications, it is common to have many instances of the JVM running simultaneously, often for lengthy periods of time. Aggregate memory requirement can therefore be large, and meeting it by adding more memory is not always an option. In these settings, overall memory footprint must be kept to a minimum. For personal computer users, the main irritation is due to long start-up time, as they expect good responsiveness from interactive applications, which in contrast is of little importance in enterprise computing.

C/C++ programs typically do not suffer from excessive memory footprint or lengthy start-up time. To a large extent, this is due to the intensive use of shared libraries. Applications share a set of shared libraries that only needs to be loaded once into memory and is simultaneously mapped into the address spaces of several running processes. This, combined with demand paging, which loads only the actually used parts of the libraries, can significantly reduce the physical memory usage. Furthermore, performance and start-up time are improved as the required data is already loaded into memory and does not need to be retrieved from disk.

The JVM itself is typically compiled as a shared library. However, the actual programs that are dynamically loaded from disk or network are stored in the private memory space of each process. This can account for a significant amount of the total memory usage. Sharing certain data structures across virtual machines may lower the memory footprint, as well as improve the performance and decrease the start-up time of applications if the format in which the data is shared requires less effort to transform into the run-time format than standard Java class files.

One approach to sharing is to execute multiple applications in the same instance of the JVM, modified to remove inter-application interference points [3], [4]. Virtually all data structures can be shared among

computations, and the resource utilization improves dramatically when compared to running multiple unmodified JVMs. However, if any application causes the virtual machine to crash or malfunction, all of the other programs executing in the faulty JVM will be affected. A different approach to sharing is demonstrated by ShMVM (for "Shared Memory Virtual Machine") [5], where certain meta-data is shared among virtual machines, each of which executes in a separate operating system (OS) process. The shared region can dynamically change, which poses various stability and robustness problems. In fact, a corruption of the shared region can lead to the cascading failures of all virtual machines participating in the sharing. The design of ShMVM required complex engineering, in part due to the aggressive choices with respect to what to share.

This work is motivated by the systems mentioned above, and in particular by the lessons learned from ShMVM. We show a design for cross-JVM meta-data sharing which is simpler, relies on existing software and OS mechanisms, and matches the performance improvements of ShMVM. At the same time, there is no loss of reliability of the underlying JVM.

The main idea is to encode Java class files as ELF shared libraries, in a parsed, verified, and mostly resolved format almost identical to the internal representation used by the JVM. Each application executes in its own virtual machine in a separate OS process, mapping such shared libraries into its address space. Efficient use of resources is accomplished by maximizing the read-only parts of the libraries, which enables cross-process sharing of read-only physical memory pages. Run-time system robustness is preserved via relying on the copy-on-write mechanism to lazily replicate process-private data. Finally, relative ease of engineering is a result of using standard shared libraries manipulation tools.

Challenges of this approach include storing meta-data outside of the regular heap, modifying the garbage collector to operate correctly on the modified heap layout, avoiding internal fragmentation in the generated shared libraries, and maximizing the amount of pre-initialization and read-only data structures in the class-encoding shared libraries. The prototype, SLVM ("Shared Library Virtual Machine"), based on the Java HotSpot™ virtual machine (or HVSM) [6], is a fully compliant JVM.

2. Overview of SLVM

This section gives an overview of using SLVM. The virtual machine can be run in two modes, (i) to generate shared libraries, or (ii) to execute applications, whenever possible using the generated libraries.

The shared libraries are generated by the SLVMGen program executed on SLVM with a *-XSLVMOuput:class* flag. The program reads in a list of names of classes and/or packages from a file and produces shared libraries according to the transformations described in Section 4. In particular, the shared libraries contain bytecodes and constant pools in a format similar to virtual machine's internal representation of these data structures, and auxilliary data structures used to quickly construct other portions of run-time class representation. Read-only and read-write sections of the shared libraries contents are separated, with the intent to maximize the amount of read-only items (Fig. 1).

The generation process consists of two steps. First, each class named either explicitly or implicitly through its package name is transformed into an object file, whenever necessary invoking native methods that interface with a library of shared object manipulation routines. Then all object files corresponding to classes of the same package are linked into a shared library, named after the package name (e.g., libjava.util.so). Lastly, class files are inserted verbatim into the object file as read-only entries. This is needed to compute certain transient load-time information and to construct string symbols. The addition of class files increases the libraries size by between 16-34%, all of which is read-only data, and thus of low impact.

At this point the libraries are ready to be used when SLVM executes applications. The information stored in the generated shared libraries describes the transformed classes completely, and is sufficient to build a runtime representation of classes during program execution.

SLVM is a fully-compliant JVM and can execute ordinary classes, whether stored as separate files or in jar archives, but the performance and memory utilization improvements are realized only when the classes are encoded as shared libraries. These can co-exist: some classes can be loaded as class files while the others can be encoded in a shared library.

To use the pre-generated shared libraries, SLVM should be executed with the *-XSLVMObjectPath:{Path to object files}* flag. Whenever a class needs to be loaded, the virtual machine tries first to locate a shared library with the required class; if such a library does not exist, regular class loading takes place. An optional *-XSLVMOnly* flag can be specified to ensure that the classes are taken from the shared libraries only. This causes SLVM to exit if any required class can not be found in the shared libraries.

Except for the optional *-XSLVM* flags described above, SLVM can be executed in the same way as HSVM. At the virtual machine and OS levels, the difference between HSVM and SLVM becomes apparent, as the use of SLVM's shared libraries moves certain meta-data outside of the heap, and the libraries themselves can be shared across multiple instances of SLVM (Fig. 2).

3. Design decisions

Several design decisions are described below, along with alternatives we considered during SLVM's

construction. They include (i) the choice of an existing shared libraries format over a potentially more flexible customized sharing protocol, (ii) the extent of sharing, (iii) the choice of shareable data structures, and (iv) the granularity of the contents of shared libraries.

3.1. Why ELF shared libraries?

ELF (Executing and Linking Format) is the standard format for storing executables and shared libraries on many UNIX platforms. It has been proven to be reliable and well-tuned. The creation of ELF formatted file is a relatively simple process because of the *libelf* library, which abstracts much of the underlying details of an ELF formatted file away.

Standard system libraries are stored in this format as it provides many crucial features that allow code to be shared. Code stored in shared libraries can be made relocatable by replacing absolute memory addresses with offsets from a global offset table for all control transfer instructions. This allows shared libraries to be mapped anywhere in an address space. Such flexibility is important to allow interoperability between shared libraries in complex systems by avoiding addressing conflicts. Data can also be stored in shared libraries, although the use of the global offset table is not necessary in order to relocate data. As the ELF format has a clear separation of read-only and write-able regions, it allows saving data-structures that are modified at runtime without affecting the read-only structures. Aside from the ability to share memory across processes, shared libraries can also be dynamically and explicitly loaded and inspected during the run-time of a program through the set of dlopen, dlsym and dlclose functions. These features satisfy SLVM's requirements for a format and functionality of cross-process data sharing. Finally, using an existing, popular, well-tested and well-tuned infrastructure has significant engineering and portability-related advantages over custom designs.

3.2. The extent of sharing

Loading of a class by a JVM is a complex and time consuming process. The file holding the binary representation of the class (the *.class* file) needs to be parsed and various values need to be computed; these values may actually be constant, yet the *.class* file format does not necessarily have that information directly available. Moreover, some of the data structures stored in *.class* files, such as bytecodes and the constant pool, undergo costly load-time modifications before reaching its initial state and run-time system format. Class loading performance affects both the start-up time of programs and the responsiveness of interactive applications. The latter is affected because class loading is performed lazily.

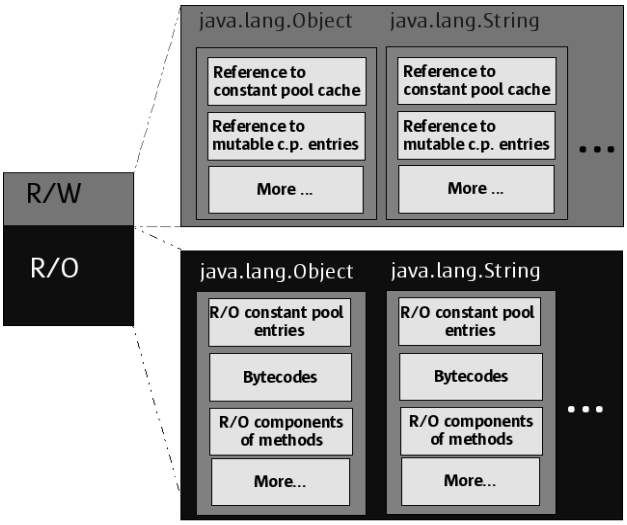

Figure 1. Separation of read-only and write-able data in *libjava.lang.so*, which encodes classes of the *java.lang* package.

Hence, a dialog box that pops up for the first time will incur the loading of one or more classes, which can cause the program to feel slow to the user. Speeding up class loading improves the user's perception of the program's performance. Class loading time can be reduced if at least some of the class-related data structures fetched from disk are already pre-initialized in the format used by the JVM.

The approach taken in SLVM is to build an initial persistent pre-formatted run-time representation of classes whose pointer values are updated during class loading. Many values that are normally calculated can now be directly loaded, which reduces the amount of effort related to class loading. It is important to note that the generated shared data structures are independent of the program execution state and of whether other (shared and non-shared alike) data structures have been loaded or not, and thus can be shared by any set of applications at any time.

3.3. Shareable data structures

Data structures shared by instances of SLVM include bytecodes, constant pools, constant pool tags, field descriptions, and constant pool cache maps. Bytecode sharing is a clear-cut choice as it is analogous to sharing compiled code normally found in shared libraries. However, HSVM rewrites bytecodes during runtime for optimization purposes, which introduces additional complexity to the design (Sec. 4.1).

The constant pool of a class holds all the constants and symbolic references of that class. The sharing of constant pools can contribute to a significant reduction in memory consumption, as it is typically the second largest data-structure following bytecodes. Many of the constant pool entries are symbolic references that are updated in place during linking and may resolve to different objects for

different programs (Section 4.3). This is problematic for sharing. A non-problematic component of the constant pool is the tag array, which stores the type information for each of the entries in the constant pool.

Constant pool caches are arrays of resolved constant pool entries of a particular type (method, field, or class). In HSVM they are write-able data structures, as the copying garbage collector can relocate the objects they point to. Constant pool caches are not shared in SLVM, as it would not have any impact on memory footprint. However, since the generation of constant pools is time consuming, SLVM stores an auxiliary read-only *constant pool map* that reduces the constant pool cache creation time.

The field descriptions uniquely describe a field and contain indices into the constant pool along with other flags pertaining to the field. These values are constants at runtime and therefore the field descriptions are potential candidates for sharing. Currently, the field descriptions are stored entirely in the write-able section of the shared library. With certain modifications, read-only information could be isolated, but the memory savings due to field-related information is minimal.

Compiled code is not encoded in shared libraries in SLVM. This is because it has been already shown that sharing of dynamically compiled HSVM code across OS processes is problematic and can actually be counterproductive [5].

3.4. Granularity of shared libraries

The contents of shared libraries determine the amount of internal fragmentation, flexibility in specifying which classes to load from a shared library, and the overhead in dynamically loading and unloading of the shared libraries.

Creating a separate shared library for each class may appear to be a reasonable choice as it maximizes the user's freedom in determining which classes should be loaded from the shared libraries. However, each class must be individually dynamically loaded, which can increase class loading time significantly as the dlopen() function must be called for each class's library. Another concern is the internal fragmentation within such single-class libraries. Each shared library, when loaded, must occupy at least one page of memory for its read-only sections and at least one page for the write-able sections (if present). Since most classes are significantly smaller than the size of a page of memory, the remaining unusable part of the page becomes wasted. Therefore, the total memory savings could be significantly reduced, if not negative, due to internal fragmentation.

Package-based shared libraries can considerably reduce the amount of dlopen() calls and the effects of internal fragmentation. Moreover, whenever a class from a package is loaded, the likelihood of loading another

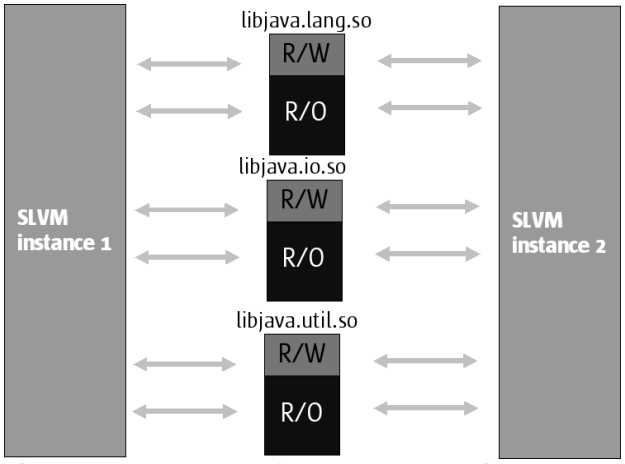

Figure 2. Instances of SLVM execute in separate OS processes, and the class-encoded shared libraries are dynamically mapped into their address spaces.

class from the same package is high. The default mode of using SLVM is thus to generate shared libraries which include all the classes of a given package; this can be customized by selecting only a subset of the classes to include.

4. Details

In order to perform safe resource sharing across virtual machines and to reduce class loading time, several issues must be addressed. They are discussed in this section, and in particular include (i) the virtual machine modifications necessary for making bytecodes read-only, (ii) separating shareable information, (iii) transformations of the constant pool, (iv) garbage collection, and (v) class loading and unloading.

4.1. Making bytecodes read-only

HSVM rewrites bytecodes internally in order to remove unnecessary checking of link resolution or class initialization status once these operations are known to have occurred, and to avoid re-computing values that remain constant after they have been computed. An example of such *quickening* is the run-time modification to the *getstatic* bytecode (it obtains the value of a static field of a given class). Its original form indexes the constant pool to compute the class, name and type of the field to fetch the value from. The first successful execution of the bytecode, which may trigger the loading and initialization of the target class, stores a reference to the target field in the constant pool cache for much faster subsequent access. To take advantage of this, HSVM internally introduces the *fast_getstatic* bytecode, which at run-time is written over any occurrence of *getstatic* upon its first execution. The *fast_getstatic* bytecode has a

similar format to that of *getstatic*, but instead of indexing the constant pool it indexes the constant pool cache. When executed, it directly obtains the reference from the constant pool cache without any additional computation or link and initialization status checks, as the entry is guaranteed to reside in the constant pool cache.

The reason why HSVM rewrites these bytecodes at runtime is due to the lazy initialization of internal run-time objects describing classes and strings. Many of these bytecodes refer to objects that have not fully been initialized. HSVM will perform this initialization at runtime after which it rewrites the bytecode to the quickened version which no longer calls the initialization method or tests the initialization status. Subsequent executions at the same bytecode index therefore will not perform any of these unnecessary operations.

In SLVM, if the *-XSLVMOuput:class* was specified, the generation of class-encoding object files is performed immediately after reaching the point in the virtual machine where the run-time representation of a class has been fully created. However, as the bytecodes have not yet been executed at this point, they are not quickened in HSVM. In SLVM they are *pre-quickened* – that is, they are replaced with fast bytecodes, so that when they are loaded in to memory as a shared library, they can be directly executed without any modifications. The bytecodes are thus read-only in SLVM's class-encoding shared libraries. Otherwise, bytecodes would have to be in write-able sections, which would lower the potential for memory footprint reduction.

In SLVM both the rewriter and interpreter are modified to handle pre-quickening. The rewriter performs pre-quickening during shared libraries generation. An example of a quickened bytecode which is interpreted differently in SLVM is *fast_getstatic*. Its interpretation in HSVM is to simply fetch the value of a field in the constant pool cache, as the cache entry is guaranteed to exist. In SLVM, because *fast_getstatic* is rewritten at initialization instead of at runtime, it has to first check whether the entry in the constant pool actually exist. If not, the interpretation falls back on the slow path and computes the entry. Subsequent executions of the bytecode are thus fast, but each of them incurs the cost of the check.

In addition to the *getstatic* bytecode, the *putstatic*, *getfield*, *putfield*, *invokevirtual*, *invokespecial*, *invokestatic*, *invokeinterface* and *ldc* bytecodes must also be pre-quickened. The interpreter must be modified in order to correctly use each of these pre-quickened bytecodes. Typically, only two additional machine instructions, load and branch, must be added in the interpretation of each of these quickened bytecodes.

In order to make the bytecode area completely read-only, two other bytecodes which the HSVM interpreter introduces at runtime are disabled. These are *fast_new* and *fast_access*. The *fast_new* bytecode, like the previously mentioned bytecodes, replaces *new* when the required class initialization is performed. This quickened bytecode is currently disabled in SLVM as it supports both *new* and *fast_new* in the interpreter, unlike the other bytecodes with quickened versions. Therefore, instead of modifying the interpreter to support a modified *fast_new*, SLVM is simply modified to not introduce *fast_new* at runtime.

The *fast_access* bytecode is rewritten in place of *aload_0* when it is followed by a *get_field* bytecode, essentially capturing the idiom of obtaining a reference to *this* object. This bytecode rewriting does not require performing any initialization, but does require knowledge of the type information of the field. The overhead of these changes is very minor as the interpreter's performance has only a small impact on overall execution time.

The HSVM's rewriter also creates the constant pool cache. The cache is built from a data structure generated to map each offset in the constant pool cache to an offset in the constant pool. Creating this map requires iterating through every entry in the constant pool, which slows down class loading. In SLVM, the map is saved into an array and exported into the object file to avoid generating it for classes loaded from a shared library.

4.2. Separating Sharable Information

The original HSVM's memory layout had to be modified for data shareable in SLVM. The bytecodes and constant pools are the largest sharable data sections for each class and are each stored in HSVM as a block of contiguous memory without a defined high level structure. Each of them directly follows an instance of a C++ abstraction class. The class contains accessor functions to retrieve data appended to its instances.

Having such instances located directly before the data section allows the class' accessor methods to reference regions of the data section simply via an offset to its *this* pointer. This removes the need for an additional pointer to the data and allows the interpreter to easily traverse between the abstraction class and the data section via offsets.

Information kept in the instances of the C++ abstraction class includes garbage collector variables, interpreter entry points and various other entries that are prone to change at runtime. It is therefore not feasible to store the abstraction classes in the read-only section of the shared library. Another reason for keeping the instances in private memory is that each contains a pointer to a C++ virtual method table (or *vtable*), typically located in the process's data segment. The abstraction class and its corresponding data sections must therefore be physically separated in memory in SLVM. Furthermore, each abstraction class must be modified to access its data section via a reference instead of via an offset.

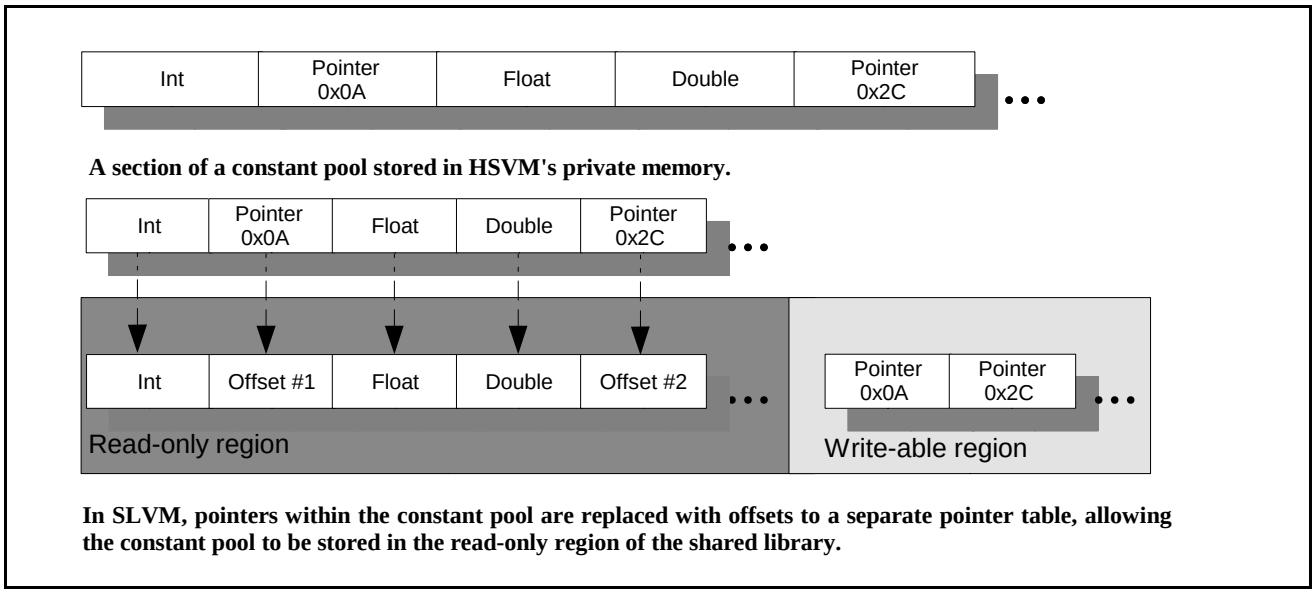

Figure 3. Constant pool organization in HSVM (top) and SLVM (bottom).

The interpreter must also be modified as it does not use the abstraction class' accessor methods to access the data section but instead directly accesses it via hard-coded offsets. All offset addition operations within the interpreter must be changed to pointer load operations. No changes are required for the dynamic compiler as it uses the abstraction class' accessor methods and the code it produces does not use any of the shared data structures.

4.3. Transformations of the constant pool

Constant pools are ideal candidate for sharing across JVM since they can be quite large. Sharing is however possible only if the original constant pool can be transformed in an data structure immutable at runtime. Class and string entries of the constant pool are problematic since they hold pointers to objects private to one JVM: the value of these pointers will be different each time the constant pool is loaded into memory as the location of the string or class will not remain the same on separate executions of the program. The pointer values can also change since the compacting garbage collector copies heap objects around. Therefore, it is not possible to simply fix the pointers to a specific value when storing the constant pool in the shared libraries.

Two solutions were evaluated: (i) replacing pointers stored in the constant pool with offsets to an array that is stored in private memory, and (ii) reordering of the constant pool. The latter requires changes to the bytecode instructions that access the constant pool in order to take the new index into account. This may change the length of the bytecode instruction itself, and in turn may require changing some control transfer bytecodes. The amount of modification this approach imposes to the JVM is unwarranted from the small gain in return. Because of this, the much less intrusive solution of replacing pointers with offsets was chosen, as it only requires modifying the behavior of the interpreter when it uses an entry in the constant pool. In SLVM, the interpretation of bytecodes that access the constant pool first fetches an offset from the constant pool and then uses it to index a private-memory array of non-constant references. This adds one load machine instruction. Figure 3 illustrates how the pointers are replaced with offsets in the constant pool.

The constant pool is also built differently during the class loading and shared library generation process: offsets are stored at places where pointers used to reside. This modification is not difficult, as the constant pool contains a tag array that describes the fundamental type of each entry stored in the constant pool. Storing any entries which the tag describes as a string or a class into the private array and replacing the entries with array offsets is sufficient to perform the conversion.

The main advantage of this solution is its simplicity. Relatively few parts of the virtual machine need to be changed in order to implement it. However, every time a pointer needs to be retrieved from the constant pool, an additional indirection must take place. Furthermore, this solution requires a read-only entry in the constant pool to store an offset in addition to the entry in the private array to store the actual pointer. However, this overhead is only in read-only memory, and is thus amortized over all instances of any application that uses the given class.

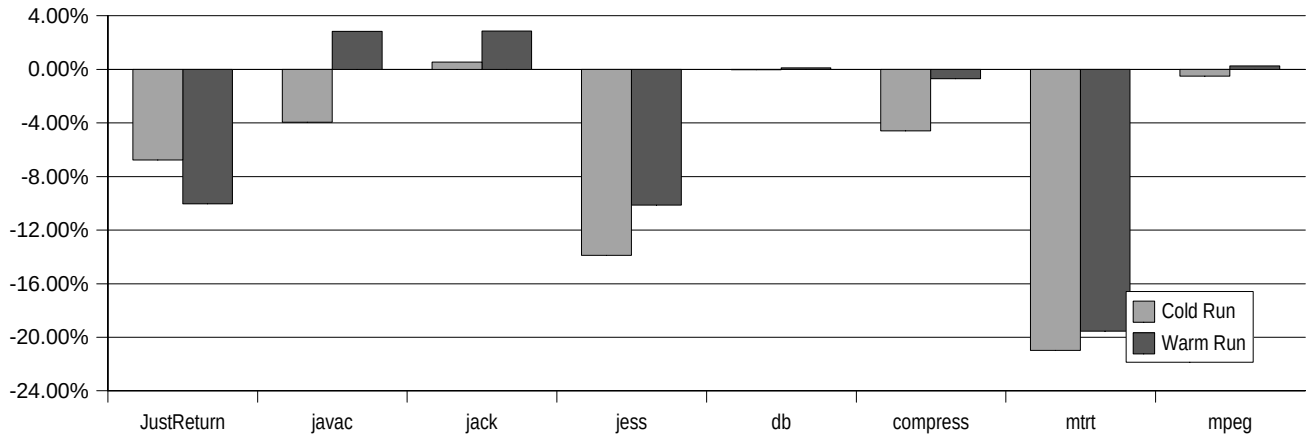

Figure 4. Performance of SLVM relative to HSVM (negative is better).

4.4. Objects outside of the heap

SLVM inherits HSVM's automatic memory management system. The heap layout is generational and the information related to classes is always stored in the *permanent generation*. Data stored there is collected very infrequently, but needs to be scanned during collections to discover live objects. The collector is copying, and it cannot be assumed that any given pointer will retain its value after a collection. In order to load a class which has been exported to a shared library, a dlopen must be called to open the shared library followed by a dlsym to read a specific symbol into memory. The memory address at which the shared information is loaded is not under our control and is not part of the heap.

The memory layout is thus changed in SLVM, as shared data structures are located outside of the heap. The *instanceKlass*, which in HSVM describes run-time class-related information, is left in the heap. Directly or indirectly it references data structures residing in shared libraries; in turn, these data structures reference other heap objects. The garbage collector was modified to properly operate on the new heap organization.

4.5. Class loading and unloading

In the Java programming language, multiple class loaders may load the same class. In such a case, if the class exists in a shared library, the library may be loaded in multiple times. A general way to make the loader load multiple instances of a library into the same process is to keep renaming the library before the load, since multiple invocations of dlopen(libA.so) will return the same handle to the only instance of libA.so loaded during the first invocation of the function. The disadvantage of this approach is that read-only segments of shared libraries are not backed by the same physical storage, as the loader does not relate the renamed libraries. The Solaris™ Operating Environment provides the dlmopen function in its run-time linker auditing interface [7], which allows for loading multiple instances of the same library such that their text (read-only) segments are backed by the same physical memory pages.

5. Performance Impact

This section discusses SLVM's performance, start-up time, and memory footprint. The experimental setup consisted of a Sun Enterprise™ 3500 server with four UltraSPARC™ II processors and 4GB of main memory running the Solaris Operating Environment version 2.8. All results are reported as relative comparisons of SLVM against the Java HotSpot virtual machine, version 1.3.1, with the Java Development Kit (JDK™) version 1.3.1 (SLVM is a modification of this code base). Complete core and application packages of classes loaded by the benchmarks were encoded as SLVM shared libraries

The benchmarks are from the SpecJVM98 suite [8] plus a very simple *JustReturn* program, which consists of only a single main method that immediately returns. For performance and start-up time measurement two kinds of measurements were performed: *cold* and *warm*. Cold execution measures the relative performance of SLVM vs. HSVM when both are executed for the first time. For SLVM, the cold execution includes the time required for the initial loading of the shared libraries. For HSVM, the cold execution includes the time needed for all core and program's class files as well as the shared libraries comprising the virtual machine to be loaded to memory from disk. The operating system's file cache was empty before cold runs. *Warm* execution is any subsequent execution of the same application, where the libraries are already loaded into the main memory. In our opinion “warm” results are more indicative of a common user experience.

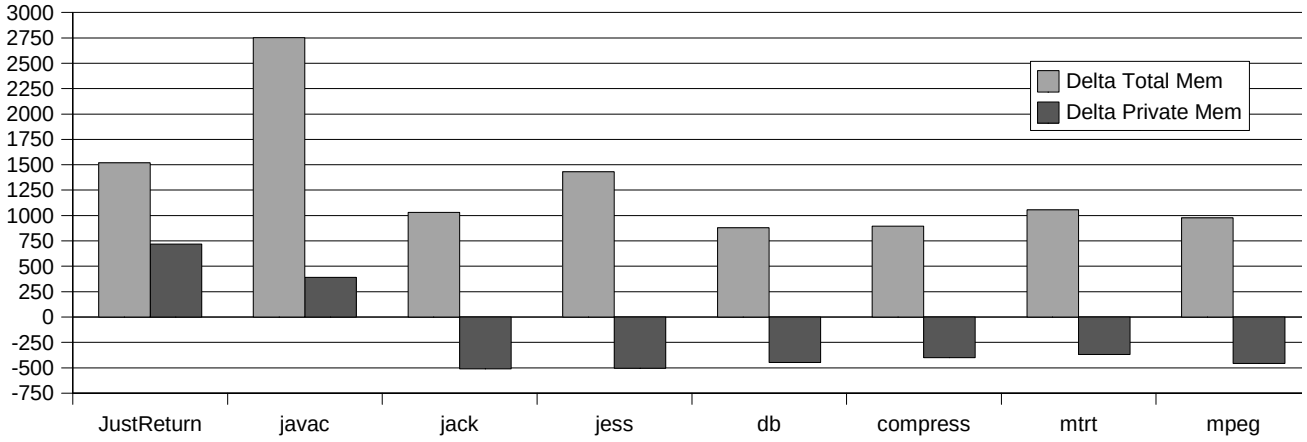

Figure 5. Memory footprint of SLVM, relative to HSVM, in kilobytes.

5.1. Execution time

Figure 4 contains the execution time of the benchmarks. For cold runs, none of the benchmarks executes slower, and most of them actually run faster with SLVM than with HSVM. Performance improvements range between 0.5% (*mpeg*) to 21% (*mtrt*). HSVM improves by a larger factor than SLVM from cold to warm executions. This explains why for warm runs SLVM does not outperform HSVM so well as for cold runs, and in fact underperforms for *jack* and *javac* by about 2.5%. *JustReturn* is an exception. Its SLVM's warm execution improves by 10% over HSVM, contrasted with 6.7% for cold runs, which indicates that the bootstrap sequence of HSVM, and consequently the virtual-machine's start-up time (Sec. 5.2) benefit more from our technique for warm executions.

The numbers are explained by the several factors. First, since less effort related to class loading is required, the applications will execute faster. A less obvious performance improving factor is revealed by timing the garbage collection. Because class meta-data does not reside in the garbage-collected heap, the amount of data structures managed by the automatic memory manager and consequently associated effort is smaller in SLVM. On the other hand, the collections are more complex.

The impact of garbage collection on the application performance time varies. With HSVM for four benchmarks it is bigger than 2% of the total execution time: jack (6.4%), *javac* (14.4%), *jess* (2.4%), and *mtrt* (6.3%). For *javac*, *db*, and *mtrt*, SLVM's modified collector increases this impact up to 1%. For other applications the overhead of the collector is smaller, by as much as 2.5% (*jess*).

5.2. Start-up time

The impact of the presented virtual machine architecture on the start-up time is determined by measuring the execution time of *JustReturn*, as most of it is spent performing start-up-related actions, including loading of bootstrap classes. The total measured process execution time is reported here as start-up time - this includes a relatively negligible virtual machine exit-related activities.

Cold start-up time improves by about 6.7%, while warm start-up time is better by about 10%. The difference is explained by the fact that cold executions spend more time fetching files from disk, so the relative performance differs less as it contains a larger similar overhead. As the original start-up time cost is in the hundreds of milliseconds, these improvements may have an observable effect for desktop users.

5.3. Memory footprint

Figure 5 shows the aggregate change in memory footprint. The amount of virtual memory required by SLVM increases (first bar in each two-bar group), but the amount needed for private memory decreases for most benchmarks (second bar), as some data previously stored in the (non-shared) permanent generation is now stored in read-only segments of shared libraries. The decrease ranges from between 300KB (*mtrt*) to almost 750KB (*mpeg*).

To illustrate how the reduced demand for private virtual memory can lower the overall memory footprint, let us take a closer look at the *compress* benchmark. When compared to HSVM, SLVM needs an additional 800KB of virtual memory more. But the private (non-shared) memory consumption actually decreases by 375KB. Thus, with three concurrent instances of *compress* the overall system requirement for physical memory is decreased by (375*3-800)KB = 325KB. Although it is hard to expect anyone to actually keep

running compress over and over again, this calculation demonstrates the potential of SLVM, as there is always sharing of at least the JDK classes between any two applications.

SLVM may actually increase the overall memory footprint, as the *javac* and *JustReturn* benchmarks indicate. In both cases the root of the problem is that relatively few classes from each package used by the application are actually loaded by the program. Since SLVM's default behavior is to encode the whole package as a shared library, much more memory (and, consequently, more private memory) is needed.

Our experiments with running the applications in 'tight' mode, where only the classes determined in advance to be loaded by the application were actually stored in the shared libraries, had virtually no impact on performance but improved the memory utilization. For instance, executing *JustReturn* causes loading of 17 class-encoding shared libraries, with the total class count of 662. However, only 184 of these classes are needed. When only the necessary classes are transformed into shared libraries, private memory requirements of *JustReturn* drop from +600KB to -50KB. However, we discourage this mode of operation unless the user has no doubts about which classes are needed by the application. Since SLVM accommodates mixed-mode loading (certain classes from disk, others from the file system/network) excluding certain classes from shared libraries is not problematic, but may negatively impact the resource utilization.

6. Discussion and related work

The lessons learned from the ShMVM project [5] had the biggest influence on our design. ShMVM enables sharing of bytecodes and compiled code among applications executing in separate virtual machines via a custom shared memory protocol. There are several major problems with that approach, though: (i) the loss of robustness due to the dynamic behavior of the shared region, (ii) only a minor performance gain yielded by the sharing of compiled code, (iii) difficulties in designing and implementing a robust and very-well performing scheme for storing and looking up shared data, and (iv) the fact that the shared region had to be mapped in each participating ShMVM virtual machine at the same virtual address. These problems led us to conclude that designs such as ShMVM are not practical.

The design of SLVM avoids all of the problems mentioned above: write-able data are automatically copied-on-write, a widely-used shared data format is taken advantage of, only the bytecodes are shared among instances of SLVM, and the addressing is flexible – shared data can be loaded anywhere in the address space of the virtual machine. In contrast to ShMVM, SLVM does not introduce any robustness degradation to the virtual machine. Finally, SLVM was much easier to engineer: it required the modification of half as many files as ShMVM did.

The performance of SLVM can be meaningfully compared to the version of ShMVM which shared class-related information across processes (i.e., did not share the dynamically compiled code). ShMVM's warm start-up time is about 85% of HSVM's (vs. 90% achieved with SLVM) but application performance is between 1-2% (relative to HSVM) better in SLVM, except for the jack benchmark, which executes about 1% slower in SLVM. The average reduction in need for private memory is very similar in both systems.

Several other projects have aimed at conserving resource of the JVM. The majority of these efforts focus on collocating applications in the same JVM, for example [3], [9], [10], [11]. The only other account of work similar to this one (apart from ShMVM) we were able to find is [12], 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. Compiled code is not shared. [12] briefly discusses these issues at a high level, without expounding on challenges and alternatives; it also does not discuss the performance of the system. That system presents another interesting feature: 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 re-used to execute another request. Thus, multiple JVMs can concurrently share resources through the shared heap, but additionally, each of them reduces start-up latency via sequential execution of applications. A follow-up system is described in [13]. Performance data presented there are promising from the perspective of reduced start-up time, but monitoring and managing the transition to the “clean slate” virtual machine can be a challenging task.

The Quicksilver quasi-static compiler [14] 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.

Incorporating a newly loaded shared library into SLVM requires less effort than Quicksilver's stitching, partly due to relying on some offset computations being done automatically by the dynamic libraries loader.

7. Conclusions

This paper discusses the design, selected implementation details, and performance of SLVM, an architecture that allows multiple instances of the Java virtual machine to share executable code. The implementation is based on an existing high-performance virtual machine, and draws from the lessons learned from our previous investigation in this area. Classes are encoded as ELF shared libraries and then loaded into the virtual machines using the standard OS mechanisms. This leads to complete separation of mutable data of virtual machines due to the copy-on-write property of shared libraries, and in consequence isolates faults of virtual machines to only the process where the fault has actually happened. Thus, the sharing does not lower the robustness of the virtual machine. Using a proven sharing format, ELF, with its extensive support tools, significantly lowered the engineering effort.

The main goal of this work is to improve several resource utilization metrics of the JVM. Lowering memory footprint was possible through maximizing the amount of read-only data structures in the shared libraries (e.g., via pre-quickening) and to storing multiple classes in the same shared library to avoid internal fragmentation. Application start-up time was decreased since several steps normally present in class loading are either eliminated (e.g., parsing) or considerably shortened (e.g., building a constant pool) in SLVM, and due to the aggressive pre-initializing of shared data to their initial values. Application performance was improved partially due to the same reasons that decrease the start-up time and in part due to the improved garbage collection behavior.

We believe that the design presented in this paper, with its combination of simplicity, robustness, use of off-the-shelf tools, and non-trivial performance, memory footprint, and startup-time improvements, is a viable approach to improving the JVM's performance and scalability when operating system process boundaries around computations are deemed necessary.

Acknowledgments. The authors are grateful to Rod Evans and Pete Soper for their help and comments. Bernard Wong participated in this project during his internship at the Sun Microsystems Laboratories in January-April 2002.

8. REFERENCES

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

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

[3] Back, G., Hsieh, W., and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. 4th OSDI, San Diego, CA, 2000.

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

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

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

[7] Sun Microsystems, Inc. *Linker and Libraries.* Available from http://docs.sun.com.

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

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

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

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

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

[13] 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.

[14] Serrano, M., Bordawekar, R., Midkiff, S., Gupta, M. Quicksilver: A Quasi-Static Compiler for Java. ACM OOPSLA'00, Minneapolis, MN, October 2002.