Safe class sharing among Java processes

Author(s):
Krause, Jens; Plattner, Bernhard

Publication Date:
2000

Permanent Link:
https://doi.org/10.3929/ethz-a-004287175

Rights / License:
In Copyright - Non-Commercial Use Permitted
Safe Class Sharing among Java Processes

Jens Krause*
IBM Research
Zurich Research Laboratory
CH-8803 Rüschlikon
jkr@zurich.ibm.com

Bernhard Plattner
Computer Engineering and Networks Lab.
Swiss Federal Inst. of Technology (ETH)
CH-8092 Zürich
plattner@tik.ee.ethz.ch

Abstract

The execution of multiple, mutually distrusting applications or multiple instances of the same application for different users in a Java Virtual Machine (JVM) requires a form of multi-processing which protects the integrity of the JVM as well as the integrity of individual applications.

Existing solutions protect processes by loading application classes in dedicated process class loaders and by allowing sharing of only the core Java classes between processes. These techniques are costly in terms of memory consumption and execution time and typically limit either the expressiveness or speed of inter-domain communication.

This paper describes a new approach which overcomes these limitations. It proposes a byte code transformation which allows the safe sharing of application classes between processes even in the presence of static fields. The feasibility of our approach is verified in a quantitative performance evaluation.

1 Introduction

In the Internet, there is a trend to execute foreign and therefore untrusted code. Examples can be found at different levels:

- Extensible Web servers with support for Java Servlets [Sun99b] allow execution of user-supplied code, e.g. to provide customized information processing functions.
- Web browsers can run one or more, remotely loaded Java Applets [Sun99a] at the same time.
- Mobile agents [CGH93] solve complex tasks in distributed systems such as network management [Y91], [FK99].
- Active networks [TSS97] aim for faster introduction of new networking protocols by defining a programming interface for network nodes.
- Java OS [Sun97] is an operating system with limited multi-process support targeting network computers.

All of the above approaches are based on Java, taking advantage of the features built into the language and the Java Virtual Machine (JVM). However, running multiple, mutually distrusting applications or multiple instances of the same application for different users in a JVM requires a form of multi-processing which protects the integrity of the JVM and the integrity of individual applications. Multi-processing support is further needed to perform resource management, i.e., to prevent a single application from exhausting the available memory, network bandwidth or storage. There are no default facilities in off-the-shelf JVMs that support these capabilities.

One way to circumvent the lack of multi-processing support in Java is to start a separate JVM for each application and to rely on the underlying operating system for those services. However, this comes

*Affiliated also to the Computer Engineering and Networks Laboratory, ETH, Zürich, Switzerland
at a cost. A JVM consumes significant amounts of memory [San99], the JVM startup time adds to the application execution time, and the communication between applications causes process context switches in the underlying operating system. Furthermore, there are small devices such as the Palm Pilot where the operating system does not support multi-processing.

Single-address-space systems [BSP+95], [EKO95], [Nel94], [WG92] use software mechanisms to provide protection. A type-safe language guarantees that references to objects cannot be forged, e.g., one cannot get hold of an object by casting an integer value into an object reference. In Java, type-safety is enforced through byte code verification, explicit casting, and type-checking.

Several projects [BG97], [TL98], [HCC+98] use Java’s type safety to provide protection for Java processes. They all suffer from the same Java characteristic: static fields, also called class variables [GJS96], have global variable semantics in Java and are accessible to all processes sharing the class in which the static field is declared. Therefore, static fields can be used to retrieve references to objects of other processes and thus to bypass process boundaries. To solve this problem, the mentioned projects propose the creation of separate class name spaces for processes. The consequences are increased memory consumption and longer application execution times, both due to separate class loading and just-in-time compilation. Further, the inter-process communication (IPC) mechanisms are either limited in their expressiveness or suffer from an overhead introduced by the use of Java’s serialization mechanism for arguments and return values.

This paper proposes an extension to the existing approaches which limits the scope of static fields in Java to the process-level without requiring the definition of separate class name spaces. The proposed solution relies on a transformation defined on the Java byte code. It reduces per-process memory requirements, speeds up process execution and provides the basis for the implementation of fast IPC mechanisms with full expressiveness.

The rest of the paper is structured as follows: Section 2 gives an overview of the existing approaches for protection introduces our new approach to protection. Section 4 discusses our implementation and Section 5 conducts a quantitative performance evaluation of the new approach. The last section summarizes and concludes the paper.

2 Protection

In this section, we first define some terminology for multi-processing within the JVM and then discuss existing approaches to protection.

2.1 Definitions

A Java process can be defined as a set of threads which is kept together by a structure called the thread group (java.lang.ThreadGroup).

The class name space of a process is defined by the class loader that loaded the initial application class, i.e., the class containing the application’s main method. A class loader’s class name space contains classes loaded by itself and all or a subset of the classes loaded by its parent class loader. For example, the class name space of an appletviewer’s applet class loader contains the applet’s classes plus the classes of the standard Java libraries loaded by the JVM’s primordial or system class loader.

A thread can only access objects which are in the object closure of the threads executing in the same thread group. The process boundary is defined by this object closure, also called the process object closure. A process’ object closure encompasses all objects created during the execution of one of the process’ threads and still referenced from the execution stack, i.e., referenced by a method local variable. Recursively, it contains all objects referenced by objects of the object closure. It contains further all objects which are referenced from static fields of classes in the process’ class name space.

For the purpose of protection, we can distinguish between static fields which are safe and such which are unsafe. Safe static fields are fields which cannot be used to leak references nor to modify the state
of a process. Constant fields (declared with modifier final) of basic types, e.g., int, float, char, and final classes, e.g., Long, Double, String are such safe static fields. An example for an safe static field is the static field Integer.MAX_VALUE.

Unsafe static fields are all non-constant fields and constant fields of more complex classes like Hashtable which are not hidden by encapsulation, i.e., static fields which are accessible from classes outside their own package or are accessible through static methods (also called class methods). For example, the static field System.out which refers to the standard output stream is unsafe because a process could close this stream thereby closing the stream for all other processes as well.

Classes containing unsafe static fields are called unsafe classes. We further call classes unsafe, if they contain methods which have side effects that have an impact on other processes. For example, the class Runtime is unsafe because of its exit() method which terminates the JVM, implicitly terminating all processes running in this JVM.

2.2 Shared class loader approach

The shared class loader (SCL) approach to protection refers to an architecture where different processes share the same class loader and thus have the same class name space.

If the class name space or part of it is shared between multiple processes, the intersection of process object closures contains those objects which are referenced from static fields of shared classes and recursively objects contained in the intersection. Objects of the intersection can be manipulated by both processes. This violates the isolation property of a process.

Figure 1 illustrates the case where two processes share the same class loader. The application class A and thus its static field A.a (of type java.lang.Object) are accessible from both processes. Therefore, the field can be used to leak object references.

![Diagram showing shared class loader approach](image)

Figure 1: The SCL approach: static field of shared class leaks object reference.

For example, assume Process 2 assigns one of its objects, object x, to the static field A.a (step 1 in Figure 1). If now Process 1 assigns the value of A.a to its instance field y.ref (step 2 in the same figure) Process 1 effectively holds a reference to an object of the object closure of Process 2. This would allow Process 1 to manipulate objects of Process 2 which potentially breaks the process’ integrity.

Holding object references that cross process boundaries causes also other problems: Which process should be accounted for the memory consumed by object x? Assume only Process 2 as the owner is accountable for it. If Process 2 runs out of memory it might want to free some memory by setting all references to object x to null and have it garbage collected. However, object x will not be garbage collected as long as object y keeps referring to it. This way Process 1 can block memory it will not be accountable for. Note that object x can not be garbage collected even when Process 2 is terminated. Accounting both processes for the object would require to update the process’ accounting information for every assignment operation.
2.3 Dedicated class loader approach

The dedicated class loader (DCL) approach as proposed by [BG97], [TL98], [HCC+98] solves the problem of static fields by dedicating a separate process class loader to each process.

In this approach, the process class loader loads all classes that implement an application and relies on the system class loader only for core Java classes. This reduces the intersection of process object closures to objects which are referenced by static fields of core JDK classes (packages java.*).

Figure 2 illustrates the DCL approach. Here, the application class A is loaded in the process class loader. Separate copies of class A, and thus of the static field A.a, exist for each process. The scope of static fields of application classes is thus limited to the process. Hence, the semantics of static fields are changed from a JVM-global variable to a process-global variable. Core Java classes containing unsafe static fields are replaced in the process class loader with safe versions.

![Diagram of DCL approach]

Figure 2: The DCL approach: Application classes are separately loaded by each process.

In summary, we can tell that the DCL approach can guarantee the integrity of processes. However, the cost is an increased memory consumption compared to the SCL approach because application classes are loaded for each process separately. If many processes are running the same application simultaneously, the usage of memory becomes significant. The SCL approach is expected to allow faster startup of additional processes with the same application because here CPU cycles for loading, resolving and eventually just-in-time compiling of classes can be saved.

3 Safe Shared Class Loader Approach

This section introduces a new approach which tries to combine the benefits of the SCL and DCL approaches. We aim to keep separate copies of the static fields while at the same time sharing classes between applications. It can be seen as a safe version of the SCL approach. Thus, we refer to it as the safe SCL approach (SSCL approach).

The idea is to split each of the original application classes into two classes: The nsp-class (class A in Figure 3), as the non-static part, contains the instance fields and all methods while the sp-class (class AStaticPart in the same figure), as the static part, contains the static fields of the original class. The static fields of the original class are transformed into instance fields in the sp-class. During runtime, exactly one instance of the sp-class is created for each process. Thus, the semantics of the static fields of the original class become those of process-global variables. Because nsp-classes and sp-classes do not contain static fields they can be shared safely (see Figure 3).

Each static field in the original class is replaced in the nsp-class with two (static) access methods. One method for read and one method for write access to the former static field which now is an instance field of the corresponding sp-class. An access method first retrieves the instance of the sp-class assigned with the current process and then reads or writes the instance variable for which it acts as an replacement.

The splitting of the original application classes causes broken references in static field accesses not only in the sp-class but also in other classes that accessed static fields of the original class. Therefore,
static field accesses in all application classes are replaced by method invocations of the corresponding access methods (see Section 4 for implementation details).

The functionality of the class initializer method of the original class is displaced into the constructor, i.e., the instance initializer method, of the sp-class. This is necessary because the transformed static fields need to be initialized for each process separately.

The Java language specification [GJS96] requires classes to be initialized on the “first active use”. Active use hereby means a method or constructor invocation, creation of an array, or access to a non-constant field of that class. Our approach guarantees these semantics only for the first started process. For subsequent processes, the static fields are initialized in the same order as for the first process already before the main() method is invoked. While an example can easily be created where our approach changes the Java semantics, it wasn’t the case for any of the tested applications in Section 5.

If we compare Figures 1, 2 and 3, we can observe that the combination of the (unsafe) SCL and the DCL approaches is achieved. The application classes are shared in the system class loader and the static fields of the original classes ended up as process-global variables with separate copies in each process.

The transformations of the SSCL approach are applied to byte code of Java class files. Thus, access to application source code is not required to deploy this approach. The transformation has been implemented using the JavaClass framework for byte code engineering [Dahm99] which allows the transformations described above directly on Java class files.

4 Implementation

After discussing the concepts of the SSCL approach in the previous section, we provide in this section details of our implementation. We first describe the transformation of normal classes. Then we show the differences for transforming interface classes and finally we discuss some special cases and limitations.

4.1 Transformation of classes

Figure 4 shows the transformation applied to classes. We can observe that static fields of the original class (A.a and A.b) show up as instance fields in the sp-class (A__staticPart.a and A__staticPart.b).

The ProcessMap A__staticPart.pm is an efficient structure which is used to retrieve the instance of the sp-class that corresponds to the current process. It is the only static field in any application class. It can not be used to bypass the process boundary because it is accessible only from within the sp-class (A__staticPart) in which it is declared (access modifier private). The sp-classes are generated by the class transformation which prevents user manipulations. The constructor (not shown in the figure) registers all instances of the sp-class with the process map. This ensures that there will be exactly one instance per process. The process map provides thus the mapping from processes to their corresponding process-global variables.
In the original class, the initialization of the static fields is done in the class initializer method A.<clinit>(). The transformation moves this functionality to the constructor method of the sp-class A_staticPart.<init>(). This is necessary because the transformed static fields need to be initialized once per process in contrast to the one time initialization during class loading. The constructor of the sp-class is declared with the private access flag to prevent abuse. It is executed if the method A_staticPart.get() cannot find an instance for the current process in the ProcessMap. This guarantees that the transformed static fields are correctly initialized before their first use. We further track the initialization order of classes and make sure that classes are initialized in the same order for all processes.

In the nsp-class, for each static field two access methods are added, e.g., the methods A._get_a() and A._set_a() replace the static field A.a. These methods are used to access the displaced fields. The access modifiers assigned to the static fields in the original class, e.g. public for A.a and private for A.b, are assigned to the corresponding access methods. For static fields that were declared to be constant (access modifier final), the _set_(a) method is left out. This guarantees the original semantics.

An example of an implementation of the access method A._get_a() is shown in Figure 4. It uses the method A_staticPart.get() to retrieve the corresponding instance of the sp-class and then selects and returns the field A_staticPart.a for which the method is a replacement. The implementation of the method A._set_a() differs only in the sense that it makes an assignment and returns a void.

The removal of the static fields needs to be reflected in all classes that reference those static fields.
Table 1: Rules for transforming classes

Figure 5 shows the necessary transformations on an example method, method B.x.O. The write access of the static field is replaced with an invocation of the access method A.set.a() and the read access is replaced with an invocation of the access method A.get.a().

In the bytecode this translates to replacing the bytecode operations GETSTATIC and PUTSTATIC with an INVOKESTATIC bytecode operation of the corresponding access method. Table 1 summarizes the transformations applied to the original class.

4.2 Transformation of interfaces

Slightly different transformations are needed for interfaces. Interface fields are implicitly declared public, static and final, i.e., one cannot reassign new values to them. For basic types, e.g., int, double etc., this means the values are constant and thus do not affect the isolation property. However, we have to assume that in general more complex field types are used, for example a Vector (field I.v in Figure 6). Such a static field could be used to leak references through process boundaries. Thus, interface fields which are always static fields also have to be moved into a sp-class. Note, the nsp-class of interfaces is also an interface, called nsp-interface, but the sp-class is a normal class.

Figure 6: Interface transformation
<table>
<thead>
<tr>
<th>Before Transformation</th>
<th>After Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Interface</td>
<td>Nsp-interface</td>
</tr>
<tr>
<td>interface name B</td>
<td>interface name B</td>
</tr>
<tr>
<td>static field c</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Additional rules for transforming interfaces

Interfaces are a special form of abstract classes. Thus interfaces can contain only method declarations but no method implementations. Therefore, we need to move the access methods into the sp-class (compare Figure 6) rather than into the nsp-interface. We do not bypass field access controls, because all interface fields are declared public, as mentioned earlier.

Figure 7 exemplifies the transformations applied to accesses to static fields of interfaces. In contrast to Figure 5, the access methods are invoked at the sp-class, e.g. I\_\_staticPart\_\_get\_i(). Table 2 summarizes the transformations that are specific to interface classes.

![Figure 7: Transformation of accesses to static fields of interfaces](image)

4.3 Special cases and limitations

Applications relying on the dynamic discovery of static fields will not find the static fields because of the transformation. One would need to modify the standard Java classes java.lang.Class, java.lang.reflect.Field and java.lang.reflect.Method to return wrappers for the displaced fields. Since this is only of benefit to a narrow class of applications, and does not carry new challenges other than implementation effort, we do not address this issue now.

The sp-classes and all of its instance fields are declared with default access which means package wide-access. This actually weakens the access control for static fields which before were declared with the private access modifier. If the java.lang.reflect package would be adapted to take into account our transformation, the reflection could be disabled for sp-classes which also resolves this problem.

5 Performance Evaluation

The evaluation carried out in this section compares the performance of our new approach, SSCL, with the DCL approach. The unsafe SCL is not considered because it is not an option for implementing multi-processing support in Java.
The goal of the proposed transformations is to reduce per-process memory consumption while maintaining the isolation property of a process. In a first approximation, one can observe that the improvements are implemented at the expense of an indirect access to process-global variables, the former static fields. Thus, we also need to quantify the impact of this indirect access with the overall application execution time.

For the performance evaluation conducted in this section, we chose a set of sample applications which provide us with realistic Java workloads:

- **CaffeineMark** [Pen99] is a widely used Java benchmark. We used the embedded version which consists of a set of low level benchmarks that allow performance comparison of JVM implementations.
- **Hashjava** version 0.8 [SBK00] is a Java library to alter symbol names directly from Java byte-code. For the performance measurements, we have it altering the symbol names of its own class files.
- **Javac** is the Java compiler as delivered with JDK 1.1.8 for Linux from IBM. For the performance measurements it compiles the source files for Jess.
- **Jess** [FH97] stands for Java Expert System Shell and is a rule engine and scripting environment. During the test, it solves the "Monkeys and Bananas" problem from the examples provided with the distribution.
- **JGL** (Java Generic Library) version 3.1 [Obj00] is an add-on for the JDK that provides a series of collections and generic algorithms. For our test, we run sequentially the array, map, and sorting benchmarks with loop parameters 12, 12, and 20, respectively.
- **Jigsaw** [War98] is an open source Web server entirely written in Java. It allows initiation of different protocol handler modules, e.g. for HTTP, FTP or remote server administration. For the measurements, we configure it to start up with the HTTP protocol handler module.
- **JLex** [Ber99] is a lexical analyzer generator. The generator is started with the included sample syntax definition.

The performance evaluation is subdivided in two sections. The first section quantifies the improvements in terms of memory consumption while the second section evaluates the impact of the SSCL approach to the application execution time. In both cases, we start with a theoretical estimation backed up with a static performance analysis before we carry out measurements on the sample applications.

The measurements are conducted on an Intel Pentium II machine (400 MHz, 128 Mbyte) running Linux, kernel version 2.2.5. The open source JVM Kaffe version 1.0.5 [Tra99] is used.

### 5.1 Memory consumption

Table 3 shows the code size of the sample applications for the original classes (before transformation) as used in the DCL approach and after the transformation for SSCL.

<table>
<thead>
<tr>
<th>Application</th>
<th>Before Transformation</th>
<th>After Transformation</th>
<th>Total</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffeine</td>
<td>14</td>
<td>33633</td>
<td>17</td>
<td>42990</td>
</tr>
<tr>
<td>hashjava</td>
<td>303</td>
<td>166757</td>
<td>64</td>
<td>38166</td>
</tr>
<tr>
<td>javac</td>
<td>628</td>
<td>593091</td>
<td>174</td>
<td>686044</td>
</tr>
<tr>
<td>Jess</td>
<td>133</td>
<td>342552</td>
<td>181</td>
<td>390995</td>
</tr>
<tr>
<td>JGL</td>
<td>45</td>
<td>79590</td>
<td>35</td>
<td>98111</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>297</td>
<td>419352</td>
<td>130</td>
<td>490656</td>
</tr>
<tr>
<td>JLex</td>
<td>96</td>
<td>126887</td>
<td>30</td>
<td>141757</td>
</tr>
</tbody>
</table>

Table 3: Code sizes of sample applications
The numbers in Table 3 represent only the classes that were actually loaded for the configured functionality of each application. The increased code sizes come mainly from the access methods, \( \_\text{get}\_xO \) and \( \_\text{set}\_xO \), that need to be added for every static field \( x \) in the original classes as described in Section 4. The overhead for the code size is between 20% and 50%. This suggests that savings can be realized under the SSCL approach if at least two processes share the same classes.

Looking at the size of class files however is a static analysis and provides only a first indication of the runtime behavior. This is because the memory consumed by the application code needs to be related to the application's runtime memory requirements which encompass the memory for the data objects and internal representations of classes, methods, and other structures.

In the following experiment, we want to quantify the runtime memory consumption of the sample applications. The reduced memory requirements should be reflected in the experiment by a higher number of simultaneously supported processes of the same application for the SSCL approach compared to the DCL approach.

The experiment is conducted as follows: We run the JVM with various fixed memory limits specified as maximum JVM heap size. For each JVM heap size, a number of processes started, all executing the same sample application. Applications that finish execution are restarted immediately within the same process. The run is successful if all applications finish execution at least once. After a successful run, we increase the number of processes by one and rerun the test. An OutOfMemoryException indicates a too high number of processes which stops the experiment. The result is the number of processes in the last successful run for the chosen memory limit. The left sides of Figures 8 to 14 show the results for the sample applications.

To better understand these results we introduce the code size to process memory ratio \( r \). We define \( r \) as the quotient of the code size \( cs \) of the original class files (see Table 3) and the amount of memory per process \( pm \) under the DCL approach. The process memory is calculated as the quotient of the maximum JVM heap size used in an experiment and the number of simultaneous processes supported under the DCL approach. For example, for javac (see Figure 10) we have \( pm = 128 \text{ MByte} / 11 \text{ processes} = 11.63 \text{ MByte} \) and \( r = cs / pm = 0.593 \text{ MByte} / 11.63 \text{ MByte} = 0.05 \). The higher \( r \) the larger is the share needed for the program code. Thus, a high \( r \) indicates a high potential for savings under the SSCL approach.

The biggest improvements can be observed for Jess (Figure 11) and JLex (Figure 14) with increases in the number of supported processes of 260% and 190%, respectively. These two applications show also the highest ratio \( r = 0.15 \) and \( r = 0.13 \), respectively).

The applications Caffeine, javac, JGL and Jigsaw also show significant improvements between 40% and 50%. Only for hashjava (Figure 9) we have almost no improvement. This can be explained with the

![Graphs showing memory consumption and execution time](image)

**Figure 8:** Caffeine: Memory consumption (left) and execution time (right).
Figure 9: `hashjava`: Memory consumption (left) and execution time (right).

Figure 10: `javac`: Memory consumption (left) and execution time (right).

Figure 11: `Jess`: Memory consumption (left) and execution time (right).
Figure 12: JGL: Memory consumption (left) and execution time (right).

Figure 13: Jigsaw: Memory consumption (left) and execution time (right).

Figure 14: JLex: Memory consumption (left) and execution time (right).
small ratio ($r = 0.02$) which doesn’t allow for relevant savings under the SSCL approach.

These results show that we achieved the initially stated goal of significantly reducing per-process memory consumption.

### 5.2 Execution time

In this section, we want to determine the impact of the SSCL approach to the application execution time. In the first experiment, we measure the access to static fields. We expect a significant slowdown for the SSCL approach as the single machine instruction to which the `GETSTATIC` operation is compiled under the DCL approach transforms to 6 method invocations, 2 type casts and a thread lookup operation to retrieve the current process and the corresponding process instance of the sp-class.

<table>
<thead>
<tr>
<th>Operation</th>
<th>time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static method invocation</td>
<td>168.0</td>
</tr>
<tr>
<td>Instance method invocation</td>
<td>175.0</td>
</tr>
<tr>
<td>Thread lookup</td>
<td>84.0</td>
</tr>
<tr>
<td>Type cast</td>
<td>137.0</td>
</tr>
<tr>
<td>Access to static field</td>
<td></td>
</tr>
<tr>
<td>- of same class (DCL)</td>
<td>2.5</td>
</tr>
<tr>
<td>- of different class (DCL)</td>
<td>55.0</td>
</tr>
<tr>
<td>- via access method (SSCL)</td>
<td>1400.0</td>
</tr>
</tbody>
</table>

Table 4: Cost of low level operations

Table 4 shows the cost we measured for relevant low level operations. We observed that Kaffe’s just-in-time compiler particularly optimizes accesses to static fields of the same class (2.5 ns versus 55 ns for access to static field of a different class). The cost of the access methods used in the SSCL approach is at about 1400 ns independent of the class to which the static field belongs. This corresponds to penalty factors of 560 (same class) and 25 (different class).

However, we can not simply extrapolate from these penalty factors and derive the overall application performance. Rather, the application’s frequency of static field accesses needs to be taken into account.

Table 5 compares the time our sample applications spent in access methods (SSCL approach) to the overall execution time. For the majority of applications, we determined overheads of less than 1% which can be neglected. Exceptions are javac and in particular JGL. Static field accesses account here for 4% and 21% of the execution time, respectively.

In the next experiment we want to quantify the implications of the different approaches for the overall application execution time. The theoretical analysis suggests advantages for the DCL approach if only one process is started because the number and size of application classes to be loaded is smaller (see Table 3) and the SSCL approach has the penalty of slower access method invocations. If more than one process is started, the SSCL approach has advantages because classes need to be loaded, resolved, and

<table>
<thead>
<tr>
<th>Application</th>
<th>Execution time [ms]</th>
<th>Access method calls number</th>
<th>Introduced overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffeine</td>
<td>24000</td>
<td>7034</td>
<td>10</td>
</tr>
<tr>
<td>hashjava</td>
<td>4000</td>
<td>12295</td>
<td>17</td>
</tr>
<tr>
<td>javac</td>
<td>14000</td>
<td>389741</td>
<td>546</td>
</tr>
<tr>
<td>Jess</td>
<td>6000</td>
<td>19115</td>
<td>27</td>
</tr>
<tr>
<td>JGL</td>
<td>4000</td>
<td>602750</td>
<td>844</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>1000</td>
<td>602</td>
<td>1</td>
</tr>
<tr>
<td>JLex</td>
<td>1000</td>
<td>3544</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5: Cost of access method invocation (SSCL) compared to overall application execution time.
just-in-time compiled only once in contrast to \( n \) times for the DCL approach.

The experiment is conducted as follows: A high priority thread starts \( n \) processes, all running the same application. The time is taken before the first process is started and after all processes are finished. The resulting difference is divided by the number of processes to obtain the average execution time per process. The maximum JVM heap size was set to 128 MBytes.

The right sides of Figure 8 to Figure 14 show our results. The experiments mostly confirm our theoretical analysis. If only one process is started, DCL performs better for hashjava, javac, JGL, and JLex (approximately 10\%). For Caffeine, Jess, and Jigsaw almost no difference can be determined. With two or more simultaneously started processes, SSCL performs better for all applications except JGL. The improvements go up to about 15\% for hashjava, javac, and Jess.

JGL is the only application which performs worse under the SSCL approach (in average about 20\%). This corresponds with the static analysis which determined an overhead introduced by the SSCL approach of 21\% (see Table 5). This overhead can not be compensated by savings related to class loading because of the tiny class size of this application.

Interesting are also the results for Caffeine (see Figure 8). The application shows high multiplexing gains in both approaches with the SSCL approach performing about 10\% better. The curves for Jigsaw (only under the SSCL approach, see Figure 13) and JLex (under both approaches, see Figure 14) show an anomaly. Here, the curves are steeper for small numbers of processes. During our measurements, we observed the garbage collector being activated with an increasing frequency as more processes are started. If 10 and more processes are started, garbage collection frequency stabilizes and the curves continue to rise monotonously, similar as for the other applications.

In summary, we can say that the amount of improvement which is achievable by introducing the SSCL approach grows both with the footprint of applications and with the numbers of separate processes that a JVM needs to support. We have shown that our approach generally leads to significant improvements. The SSCL approach performs worse, if the application makes exhaustive use of static fields.

6 Summary and Conclusion

In this paper, we presented a new approach for multi-processing in Java. This new approach allows to safely share classes between protection domains. In a quantitative performance evaluation, we showed that our approach can help to reduce per-process memory consumption and application execution time.

We can see benefits of the new approach to applications on both sides of the client-server programming model. Server-side applications can run for different users with different privileges and nevertheless share the application code. On the client side, different applications can run simultaneously in the same JVM without interfering with each other. The proposed savings can be realized if different applications use the same library, e.g. a graphics or algorithmic package.

Fast IPC mechanisms in existing Java multi-processing approaches \( \text{[HCC}^+98] \), \( \text{[BTS}^+98] \) limit argument types to classes which can be shared safely without leaking references. Therefore, those mechanisms are typically limited in their expressiveness. Combined with the SSCL approach, fast IPC mechanisms can be implemented which provide full expressiveness.

A limitation of the current implementation is that the byte code transformation is not transparent when the reflection API is applied to static fields. However, we provided indications for modifications to the reflection API implementation to resolve this issue.

References


