Safe Class Sharing among Java Processes

Jens Krause
IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
jkr@zurich.ibm.com

Bernhard Plattner
ETH Zürich, TIK
Gloriastrasse 35
8092 Zürich
Switzerland
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, startup time and 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 [Sun99a] 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 [Sun99b] at the same time.
- Mobile agents [CGH+95] solve complex tasks in distributed systems such as network management [YGY91], [FKK99].
- Active networks [TSS+97] aim for faster introduction of new networking protocols by defining a programming interface for network nodes.
- JavaOS [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 at a cost. A JVM consumes significant amounts of memory [SCO99], the JVM startup time adds to the application startup 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], [Nels91], [WiGu92] 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 [BaGo97], [TuLe98], [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 startup times, both due to separate class loading and just-in-time compilation. Further, the inter-process communication (IPC) mechanisms 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 startup and provides faster IPC.

The rest of the paper is structured as follows: Section 2 gives an overview of the existing approaches for protection and section 3 discusses implications for IPC. The next section introduces our new approach to protection. Section 5 discusses implementation details and Section 6 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 applications `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 application class loader could contain application classes plus the classes of the core Java libraries loaded by the JVM's 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 The shared class loader approach

The *shared class loader approach* (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 can violate the integrity 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.

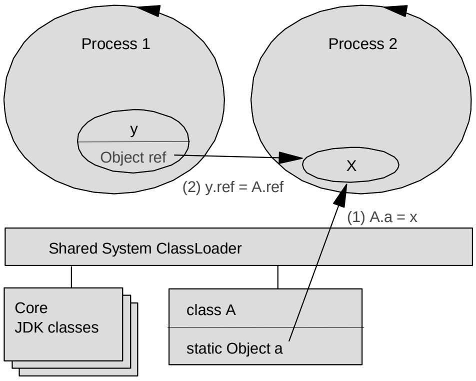


Figure 1: The SCL approach: static field of shared class leaks object reference.

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 collect 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 The dedicated class loader approach

The dedicated class loader approach (DCL approach) as proposed by [BaGo97], [TuLe98], [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 so limited to the process. Hence, the semantics of static fields are changed from a JVM-global variable to a process-global variable. Core JVM classes containing unsafe static fields are replaced in the process class loader with safe versions.

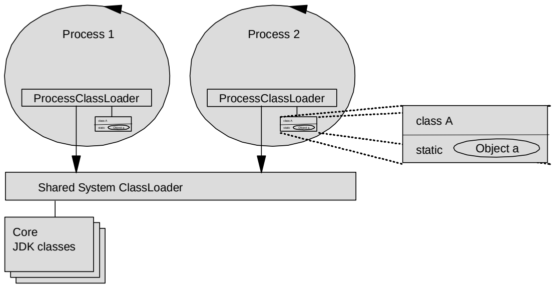


Figure 2: The DCL approach: Application classes are separately loaded by each process.

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 also 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 Inter-Process Communication

This section discusses the implication of the SCL and DCL approaches to IPC. The simplest and fastest way to communicate with another process in Java would be through direct method invocation on a server object of the corresponding process. However, this requires a reference to the server object which crosses the process boundary and which in turn is a potential threat to the integrity of a process.

Object references can safely be shared among processes by means of revocable *capabilities*. A capability encapsulates a references to a server object (see Figure 3). IPC is only possible with objects for which a capability is exported by the owner process.

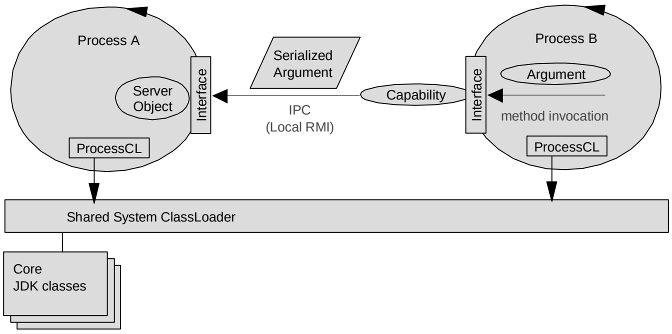


Figure 3: Schema for IPC via revocable capabilities.

A capability implements all or a subset of the interfaces of the referenced server object. It can thus provide the same functionality to clients in other processes as the referenced server object provides to process-internal clients. By exporting a capability that implements only a subset of the interfaces the functionality provided to external clients can be limited.

By keeping track of exported capabilities, it is possible to efficiently analyze inter-process dependencies. On process termination all exported capabilities are revoked, i.e. the reference is set to null. This guarantees that no external references exist and the process memory as well as other resource represented by the process’ objects can be released.

Method invocations on capabilities have semantics which differ from those of normal objects. Because the invocation crosses the process boundary, a capability has to make sure not to leak direct object references, neither from the client, the caller, nor from the server, the callee. Therefore, the capability passes arguments to IPC method invocations by copy. The same is true for return values and exceptions. The copy is actually a deep copy meaning objects referenced by an argument object and thus the object closure of an argument object are copied.

Capabilities experience a special treatment and are passed by shallow copy. This means that the capability object is copied but not the server object it is referring to. The rules are applied recursively such that capabilities contained in the closure of a normal object are also passed by shallow copy. The semantics correspond to those defined by the Java remote method invocation (RMI) [Sun98]. IPC thus can be seen as a *local RMI* within the same JVM.

The *default mechanism* for copying arguments between processes uses Java’s object serialization. The argument is serialized in a byte array which is then sent over to the other process where it is deserialized thereby creating a copy of the original argument. For objects whose classes are shared between processes, a *fast copy mechanism* can be implemented in native code which avoids the intermediate byte array representation by directly copying instance fields.

For the DCL approach, the fast copy mechanism is limited to the core JDK classes because only those are shared between processes. [HCC+98] allows explicit sharing of argument classes to enable fast copying. However, those classes must not contain static fields nor must they refer to classes containing static fields. Hence, this significantly limits the expressiveness of fast copy IPC in the DCL approach. The SCL approach does not suffer from this limitation because here all classes are shared.

4 The Safe Shared Class Loader (SSCL) Approach

In Section 2, we have seen two approaches which provide protection in a multi-process JVM. The SCL approach is unsafe because static fields of application classes become JVM-global variables. Those fields can be used to bypass process boundaries and thus violate process isolation.

The DCL approach loads application classes in separate per-process class loaders. This creates separate copies of the application classes and thus separate copies of the static fields. In the DCL approach, the semantics of static fields are those of process-global variables. However, the repeated class loading in the DCL approach introduces a significant overhead as discussed in Section 2.3.

Our new approach tries to combine the benefits of the previous two approaches. We aim to keep separate copies of the static fields while at 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 4), as the non-static part, contains the instance fields and all methods while the *sp-class* (class `A_staticPart` 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 securely (see Figure 4).

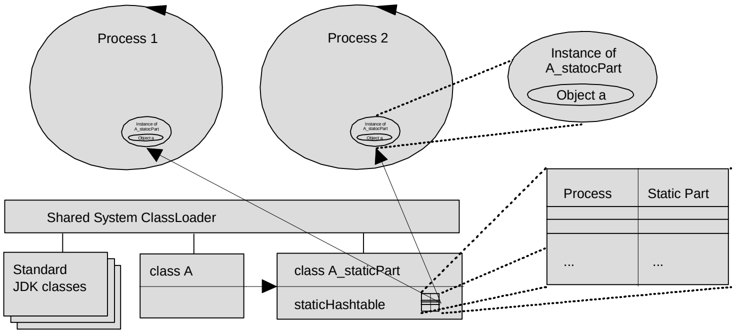


Figure 4: Process-global variables

Each static field in the original class is replaced in the nsp-class with two (static) access methods, one for read and one 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 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 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 5 for implementation details).

If we compare Figures 1, 2 and 4, 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.

As mentioned in Section 3, fast copy IPC is limited in the DCL approach to argument types of the core JDK classes because only those are shared between processes. This limitation does not apply to the SSCL approach where all classes are shared. In the SSCL approach, the fast copy mechanism can be used for arbitrary argument types.

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 transformations described above directly on Java class files.

5 Implemention of an SSCL

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

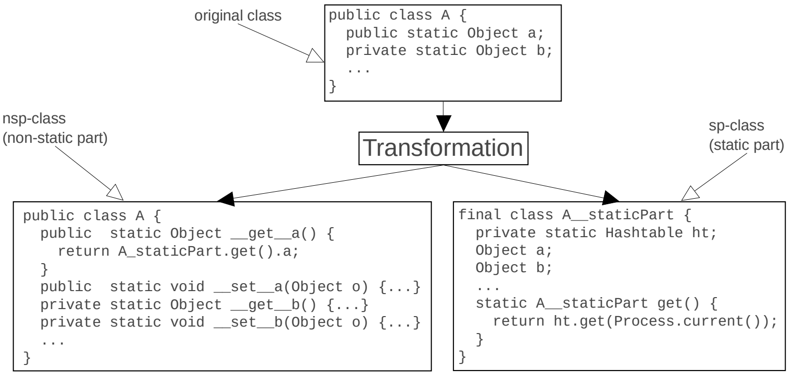
5.1 Transformation of classes

Figure 5 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 hash table `A__staticPart.ht` 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 hash table. This ensures that there will be exactly one instance per process. The hash table 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 hash table. This guarantees that the transformed static fields are correctly initialized before their first use.

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__`*() method is left out. This guarantees the original semantics.

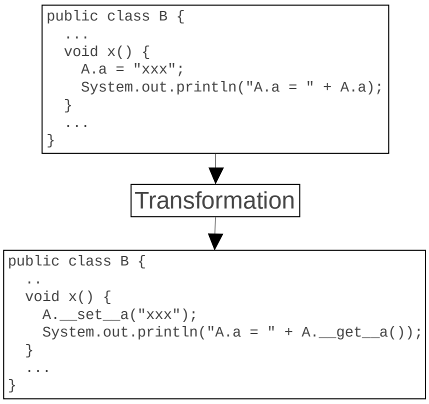


```
public class A {
  public static Object a;
  private static Object b;
  ...
}
```

```
public class A {
  public  static Object __get__a() {
    return A_staticPart.get().a;
  }
  public  static void __set__a(Object o) {...}
  private static Object __get__b() {...}
  private static void __set__b(Object o) {...}
  ...
}
```

```
final class A__staticPart {
  private static Hashtable ht;
  Object a;
  Object b;
  ...
  static A__staticPart get() {
    return ht.get(Process.current());
  }
}
```

Figure 5: Class transformation



```
public class B {
  ...
  void x() {
    A.a = "xxx";
    System.out.println("A.a = " + A.a);
  }
  ...
}
```

```
public class B {
  ..
  void x() {
    A.__set__a("xxx");
    System.out.println("A.a = " + A.__get__a());
  }
  ...
}
```

Figure 6: Transformation of accesses to static fields

An example of an implementation of the access method `A.__get__a()` is shown in Figure 5. 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 access the static fields. Figure 6 shows the necessary transformations on an example method, method `B.x()`. 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()`.

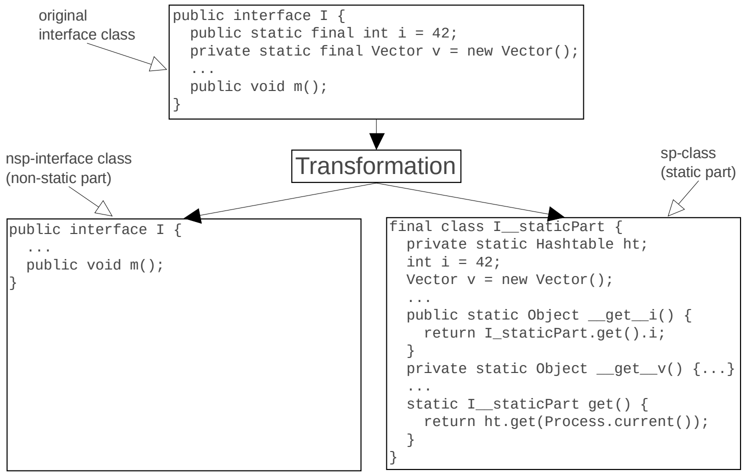
Before Transformation	After Transformation	
Original class	Nsp-class	Sp-class
class name A	class name A	class name A__staticPart
static field a	static methods __get__a() and __set__a()	instance field a
field access modifiers	method access modifiers	field access modifier
- public, protected, private	- public, protected, private	- n/a
- final, transient, volatile	- n/a	- final, transient, volatile
class initialization method <clinit>()	n/a	constructor <init>()
access to static field within a method	method invocation	n/a
- GETSTATIC A.a	iINVOKESTATIC __get__a()	
- PUTSTATIC A.a	INVOKESTATIC __set__a()	

Table 1: Rules for transforming classes

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

5.2 Transformation of interfaces

Slightly different transformations are needed for interfaces. Interface fields are implicitly declared `public`, `static` and `final`, i.e., one can not 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 7). Such a static field could be used to exchange references across process boundaries. Thus interface fields, which are always static fields, also have to be moved into an sp-class. Note, for interfaces the nsp-class is also an interface, called *nsp-interface*, but the sp-class is a normal class.

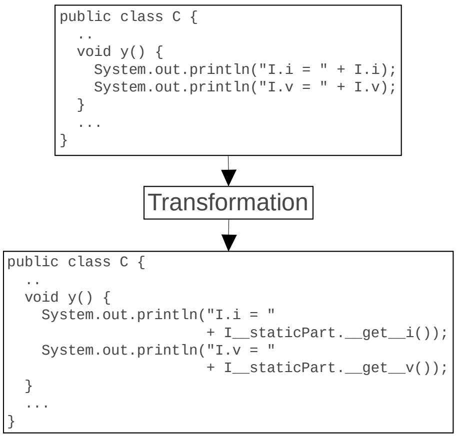


```
public interface I {
  public static final int i = 42;
  private static final Vector v = new Vector();
  ...
  public void m();
}
```

```
public interface I {
  ...
  public void m();
}
```

```
final class I__staticPart {
  private static Hashtable ht;
  int i = 42;
  Vector v = new Vector();
  ...
  public static Object __get__i() {
    return I_staticPart.get().i;
  }
  private static Object __get__v() {...}
  ...
  static I__staticPart get() {
    return ht.get(Process.current());
  }
}
```

Figure 7: Transformation of interfaces



```
public class C {
  ..
  void y() {
    System.out.println("I.i = " + I.i);
    System.out.println("I.v = " + I.v);
  }
  ...
}
```

```
public class C {
  ..
  void y() {
    System.out.println("I.i = "
                       + I__staticPart.__get__i());
    System.out.println("I.v = "
                       + I__staticPart.__get__v());
  }
  ...
}
```

Figure 8: Transformation of accesses to static fields of interfaces

The problem that appears here is that interfaces can contain only method declarations but no method implementations. Therefore, we need to move the access methods into the sp-class (compare Figure 7) rather than into the nsp-interface. We do not bypass field access controls, because all interface fields are declared `public`, as mentioned before.

Figure 8 exemplifies the transformations applied to accesses to static fields of interfaces. In contrast to Figure 6, 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.

Before Transformation	After Transformation	
Original Interface	Nsp-interface	Sp-class
interface name B	interface name B	class name B__staticPart
static field c		instance field c plus static method __get__c()

Table 2: Additional rules for transforming interfaces

5.3 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 resolves also this problem.

At the beginning of this section, we mentioned that the implementation of the class initialization method `<clinit>()` has been transformed into the constructor or instance initialization of the sp-class. In cases where the class initialization method has also side effects on static fields of other classes it might be that the class initialization does not happen in the usual order or not at all. To prevent this, we make sure that all classes whose `<clinit>()` are more complex than just initializing static fields with constant values are executed on process startup in the same order as it would happen for the original classes.

6 Performance Evaluation

The goal of the proposed transformations is to reduce per-process memory consumption, process startup time, and the cost for inter-process communication while maintaining the isolation property of a process. In a first approximation, one can observe that the improvements are implemented on the expense of an indirection in the access to process-global variables, the former static fields. In order to evaluate the proposed solution quantitatively, this section conducts a performance evaluation.

For the performance evaluation, we chose three sample applications which provide us with realistic Java workloads:

- Jigsaw [W3C98] 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 configured it to start only with the HTTP protocol handler module.
- Jess [Frie97] stands for Java Expert System Shell and is a rule engine and scripting environment. During the test, it solved the “Monkeys and Bananas” problem from the examples provided with the distribution.
- CaffeineMark [Pend99] is a widely used Java benchmark. We used the embedded version which consist of a a set of low level benchmarks that allow to compare the performance of JVM implementations.

The performance evaluation is subdivided in two sections. The first section, measures the low level operations access to static fields and IPC. In the following section, our sample applications are run to provide a quantitative estimate of the improvements that can be expected using the proposed solution.

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 beta 4 [Tran99] is used.

6.1 Performance of low level operations

In the first experiment, we measured the access to static fields. We expect a significant slowdown for the SSCL approach because the single machine instruction to which the `GETSTATIC` operation can be compiled in the DCL approach versus a 6 method invocations, 2 type casts and a thread lookup operation which are necessary to retrieve the current process and the corresponding process instance of the sp-class in the SSCL approach.

Operation	time [ns]
Static method invocation	168.0
Instance method invocation	175.0
Thread lookup	84.0
Type cast	137.0
Access to static field	
- of same class (DCL)	2.5
- of different class (DCL)	55.0
- via access method (SSCL)	1400.0

Table 3: Cost of low level operations

Table 3 shows the cost 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 for access to static fields of 560 (same class) and 25 (different class).

However, we argue that static field accesses are infrequent and thus do not significantly reduce the overall performance of an application. Table 4 compares for our sample applications the time spent in access methods (SSCL approach) with the overall execution time. We determined overheads of less than 1 percent for all application which can be neglected.

As an optimization, the instance of the sp-class containing the former static field could be buffered in a local method variable on the stack. This would provide a shortcut which can be expected to perform as good as the direct access to a static field. However, as Table 4 shows, we did not find an application where this performance penalty is significant and thus postponed the implementation.

Application	Application runtime [msec]	Access method invocations Number	Time [ms]	Introduced overhead
Jigsaw	2,000	535	0.75	0.04%
Jess	12,000	19,079	26.71	0.22%
Caffeine	24,000	5,417	7.58	0.03%

Table 4: Cost of access method invocation (SSCL) compared to overall application execution time.

The second experiment determines the cost of IPC. Here, we compare two versions of local RMI: one using Java's object serialization which is the default case for the DCL approach and a fast copy mechanism which is used in the SSCL approach. As the name suggests, we expect better performance for the fast copy mechanism because it directly copies objects instead of using a intermediate byte code representation.

In Table 5, we can observe speedup factors of about one to two orders of magnitude when the fast copy mechanism is used. The speedup factor increases with the increasing argument size. Similar speedup factors where also published in [HCC98].

Argument	Byte Array		String	
Size[byte]	serialized	fast copy	serialized	fast copy
1	2.00	0.15	0.23	0.06
10	2.10	0.15	0.28	0.06
100	2.40	0.16	0.69	0.06
1,000	6.50	0.17	4.73	0.08
10,000	42.50	0.30	46.49	0.31

Table 5: Cost of IPC (times [ms])

6.2 Scaleability and performance of sample application

The goal of this section is to provide a quantitative estimate of the improvement in terms of memory consumption and startup time that can be achieved using the SSCL approach.

Table 6 lists the sample application that were used. Jigsaw [W3C98] 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 configured it to start only with the HTTP protocol handler module. Jess, the Java Expert System Shell, is a rule engine and scripting environment [Frie97]. During the test, it solved the "Monkeys and Bananas" problem from the examples provided with the distribution. The third application in the table is the embedded version of the widely used CaffeineMark Java benchmark [Pend99]. It consist of a a set of low level benchmarks that allow to compare the performance of JVM implementations.

	Before Transformation			After Transformation			
	Original classes		Static	Nsp-classes		Sp-classes	
Application	Size [byte]	Number	fields	Size [byte]	Number	Size [byte]	Number
Jigsaw	450,449	142	304	527,534	142	70,540	44
Jess	350,825	179	109	400,880	179	19,985	21
Caffeine	13,715	14	14	16,899	14	2,030	2

Table 6: Sample applications

The numbers in Table 6 represent only the classes that were actually loaded for the configured functionality of each application. The increased class size comes mainly form the access methods, `__get__x()` and `__set_x()`, that need to be added for every static field x in the original classes as described in Section 5. The overhead for the class size is between 20% and 40%. This means if the classes are shared between at least two processes memory savings can be realized.

Looking at the size of class files however is a static analysis and provides only a first indication. This is because the memory consumed by the application code needs to be related to the applications overall memory needs which include also the memory for data objects the application is working with.

The goal of the first experiment in this section is to quantify the overall memory consumption. We expect to confirm the results of the static analysis above which should be reflected in the experiment by a higher number of simultaneous supported processes of the same application for the SSCL approach.

The experiment is conducted as follows: We run the JVM with a fixed memory limit specified as the JVM heap size. Then, a set of processes were sequentially started, all executing the same sample application, until one process ran out of memory indicated by an exception. The startup of a new process is delayed until one can be sure that all existing processes are completely loaded and were executing long enough to have allocated their required share of memory. Application executions that finished were restarted within the same process. This prevents the garbage collector from freeing a process' memory. The same experiment was conducted with both approaches, DCL and SSCL.

The result of the experiment represents the maximum number of processes of a given application that can be simultaneously supported for a given JVM memory limit. Figures 9, 11 and 13 show the results for the sample applications. We can observe improvements for the SSCL approach of about 50% for Jigsaw and Caffeine, and an improvement of 400% for Jess. Jess benefits more from the SSCL approach because the ratio of memory required for data versus code is less than for Jigsaw or Caffeine.

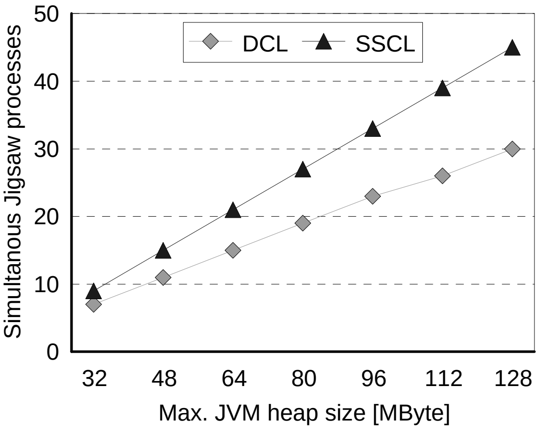


Figure 9: Max. number of simultanous Jigsaw processes.

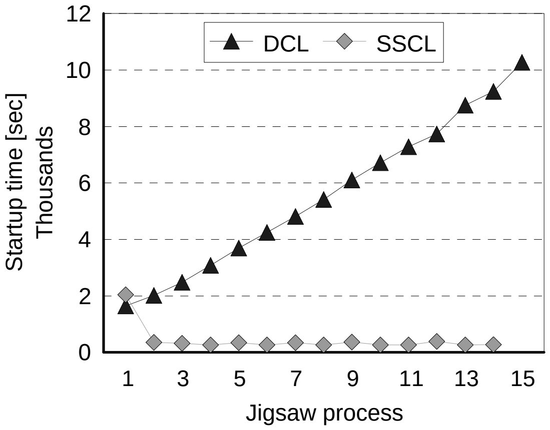


Figure 10: Startup time for n-th Jigsaw process

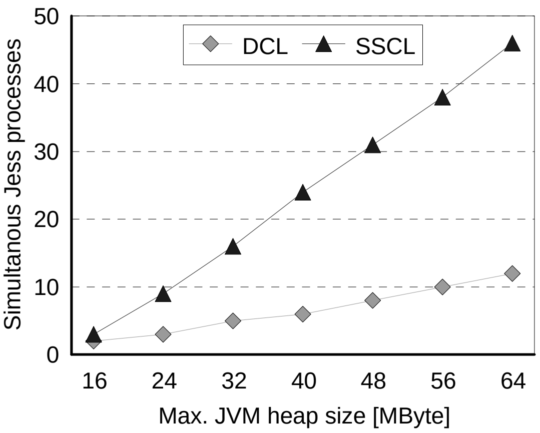


Figure 11: Max. number of simultanous Jess processes.

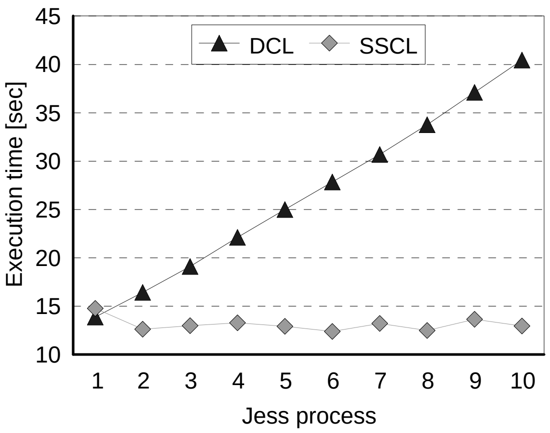


Figure 12: Execution time for n-th Jess process

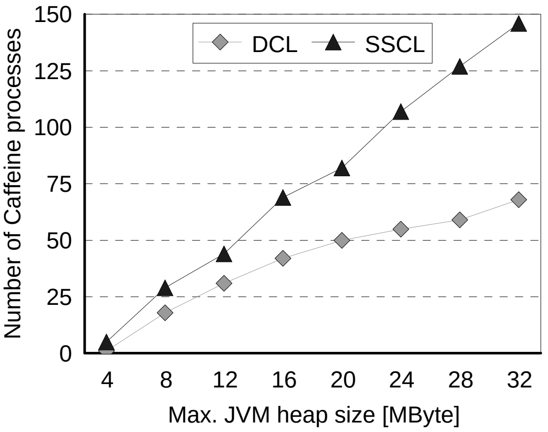


Figure 13: Max. number of simultanous Caffeine processes

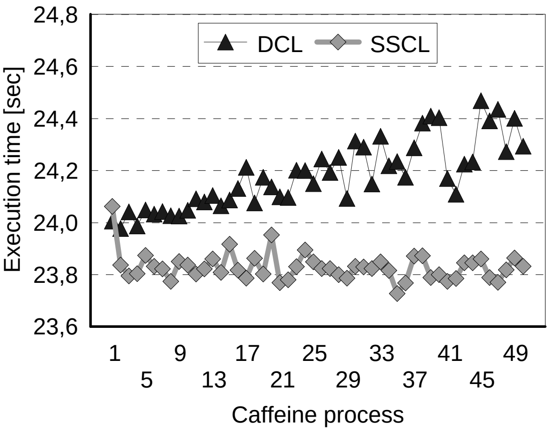


Figure 14: Execution time for n-th Caffeine process

In the second experiment, we want to quantify the implications of the different approaches for the startup or 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. If more than one process is started, the SSCL approach should perform better because classes need to be loaded only once in contrast to n times for the DCL approach.

The experiment was conducted in a similar way as the previous. A new process was started only after the previous process had finished execution. Finished processes were not restarted but a reference was kept to prevent garbage collection of classes. The limit for the JVM heap size was set to 128 Mbyte.

Figures 10, 12 and 14 confirm the theoretical analysis. For the SSCL approach, the execution time of the first process is about 2 seconds higher for Jigsaw and Jess. The difference is less significant for Caffeine which can be attributed to the smaller footprint of the application (10 times fewer classes).

The linear increase in startup times with the conventional class loader in Figures 10, 12 and 14 seems mildly peculiar, because these graphs do not depict aggregated times (a flat curve was expected). Although we have confirmed this behavior with several popular JVM implementations (JDK 1.1.7 for Linux based on Sun SDK, JDK 1.1.8 for IBM AIX, Kaffe for Linux) we cannot pinpoint the exact reason, which must relate to some common denominator in how these implementations handle data structures. Even without this additional boost (which may disappear in other conventional implementations), the SSCL approach consistently outdoes its conventional counterpart for n > 1, because its curves falls below the others' after incurring specific overheads each once for n = 1.

The amount of improvement that is available by introducing the SSCL approach grows both with the footprint of applications (Figures 9, 11, and 13) and with the numbers of separate processes that a JVM needs to support (Figures 10, 12, and 14). We have shown that our approach leads to significant improvements with a set of popular and unmodified Java workloads.

7 Summary and Conclusion

In this paper we presented a new approach for multi-processing in Java. This new approach allows for safe class sharing between protection domains. In a quantitative performance evaluation we showed that our approach can help to reduce per-process memory consumption, process startup time, and delay for inter-process communication.

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.

As a side result of the performance evaluation, we discovered that class loading performance of different JVM implementations decreases with an increasing number of classes already loaded. This suggests that there is space for optimization of currently deployed class loading mechanism. At the same time, it provides an additional argument for the new approach which reduces the number of loaded classes significantly when the same application is run multiple times in parallel.

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

[BaGo97] Dirk Balfanz and Li Gong: *Experience with Secure Multi-Processing in Java*, Princeton University, Technical Report 560-97, September, 1997.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak et al: Extensibility, Safety, and Performance in the SPIN Operating System. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.

[CGH+95] D. Chess, B. Grossof, C. Harisson et al.: Itenerant Agents for Mobile Computing. IEEE Personal Communications, October 1995.

[Dahm99] Markus Dahm: *Byte Code Engineering*, to appear in Proceedings of Java-Informations-Tage’99 (JIT'99), Düsseldorf, 1999.

[EKO95] R.Engler, M. Kaashoek, J. James O’Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. 15th ACM Symposium on Operating Systems Principles, 1995

[Frie97] Ernest J. Friedman-Hill: Jess, The Java Expert System Shell. Technical Report SAND98-8206,Distributed Computing Systems,Sandia National Laboratories, Livermore, CA, 1997. http://herzberg.ca.sandia.gov/jess

[FKK99] M. Feridun, W. Kasteleijn, J, Krause: Distributed Management with Mobile Components. In Proceedings of International Symposium on Integrated Network Management, Boston, MA, 1999.

[GJS96] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification*. Edition 1.0, 1996.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken: *Implementing Multiple Protection Domains in Java*, 1998 Usenix Annual Technical Conference.

[Nels91] G. Nelson, ed. *System Programming in Modula-3*. Prentis Hall, 1991.

[Pend99] Pendragon Software Corporation: CaffeineMark 3.0
http://www.pendragon-software.com/pendragon/cm3/index.html

[SCO99] Santa Cruz Operation, Inc.: *Enabling Java technology to efficiently support multi-user server applications*, Technical White Paper, 1999.

[Sun97] Sun Microsystems, Inc.: JavaOS: A Standalone Java Environment, White Paper, 1997

[Sun98] Sun Microsystems, Inc.: Java Remote Method Invocation (RMI) Specification. 1998 http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

[Sun99a] Sun Microsystems: Java Servlet API Specification. Version 2.2, November 1999. http://java.sun.com/products/servlet/2.2/

[Sun99b] Sun Microsystems: Applets Home Page. http://java.sun.com/applets/

[Tran99] TransVirtual Technologies, Inc.: *Kaffe - one for all or to have a free Java*. 1999. http://www.transvirtual.com/one4all/

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie et al.: A Surwey of Active Network Research. IEEE Communications Magazine, Januar 1997

[TuLe98] Patrick Tullmann and Jay Lepreau: *Nested Java Processes: OS Structure for Mobile Code*, Proceedings of the Eighth ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.

[W3C98] World Wide Web Consortium: *Jigsaw Activity Statement*, 1998. http://www.w3.org/Jigsaw/Activity.html

[WiGu92] N.Wirth, J.Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Adison Wesley, 1992.

[YGY91] Yechiam Yemini, Germán Goldszmidt, and Shaula Yemini. *Network Management by Delegation*. In The Second International Symposium on Integrated Network Management, Washington, DC, April 1991.