Application Isolation in the Java™ Virtual Machine

Grzegorz Czajkowski
Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043, USA
+1-650-336-6501

Grzegorz.Czajkowski@sun.com

Abstract

To date, systems offering multitasking for the Java™ programming language either use one process or one class loader for each application. Both approaches are unsatisfactory. Using operating system processes is expensive, scales poorly and does not fully exploit the protection features inherent in a safe language. Class loaders replicate application code, obscure the type system, and non-uniformly treat 'trusted' and 'untrusted' classes, which leads to subtle, but nevertheless, potentially harmful forms of undesirable inter-application interaction.

In this paper we propose a novel, simple yet powerful solution. The new model improves on existing designs in terms of resource utilization while offering strong isolation among applications. The approach is applicable both on high-end servers and on small devices. The main idea is to maintain only one copy of every class, regardless of how many applications use it. Classes are transparently and automatically modified, so that each application has a separate copy of its static fields. Two prototypes are described and selected performance data is analyzed. Various aspects of the proposed architectural changes to the Java Virtual Machine are discussed.

Keywords

Java Virtual Machine, multitasking, application isolation.

1 INTRODUCTION

The growing popularity of the Java programming language [2] brings about an increased need for executing multiple applications written in the language and co-located on the same computer [4]. Ideally, such applications should be protected from one another, which means that an application should not be able to corrupt the data of another and should not be able to prevent another application from performing its activities. At the same time, marginal system resources needed to start new applications should be as small as possible so that the number of concurrently executing applications can be as high as possible.

One approach to multitasking in the Java programming language is to start each application in a separate copy of the Java Virtual Machine (JVM™) [21]. This typically requires spawning a new operating system process for each application and provides strong separation between applications but uses large amounts of resources (memory, CPU time) and makes inter-application communication expensive.

An alternative is to execute applications in the same instance of the JVM. Typically, each application is loaded by a separate class loader [20]. This code replication is especially wasteful in the presence of just-in-time compilers (JITs). Current JVM implementations separately compile and separately store the JITed code of each loaded class, regardless of whether the class has already been loaded by another application or not. This can easily lead to significant memory footprints since, as [8] indicates, on the average, a byte of bytecode translates into five to six bytes of native code. Combined with the safety of the language, this approach leads to systems where applications are *mostly* isolated from one another. The place where the isolation breaks is the interaction of applications through static fields and static synchronized methods of system classes (they are not subject to per-application replication).

Two current trends make us question the future usefulness of these approaches. On one end of the computing power spectrum, high-end high-throughput servers have to deal with large numbers of concurrently executing programs written in the Java programming language. Increasingly, in addition to traditional, large and self-contained applications, other entities, such as applets, servlets [18], and Enterprise JavaBeans™ components [17], enter the picture. The process-based approach is unacceptable in these settings, as it allocates large amounts of system resources to starting many copies of the JVM and thus scales very poorly. Using class loaders has the potential to scale better but typically resources are wasted on replicating application code when more than one application executes the same class. Isolation inconsistencies pointed out earlier make this approach unsafe in general.

On the other end of the spectrum, small-footprint JVMs, targeting small devices, are emerging. They typically lack many features available in full implementations of the JVM. An example is the K Virtual Machine (KVM) [19]. Since the KVM specification does not require that its implementations provide class loaders, multitasking in a single instance of the KVM is possible only when all applications are trusted and guaranteed not to interfere with one another. Process-based multitasking using the KVM is also problematic since it is meant for small devices, which do not necessarily have an OS or a process model with adequately strong application separation guarantees.

The goal and contribution of this paper is to address the central problem of multitasking: isolating (protecting) applications from

one another. The approach presented here advocates sharing all classes among all applications in the same instance of the JVM. The only non-shared entities are certain static fields and some monitors.

The idea has been implemented in two ways. The first prototype, based on bytecode editing, has certain limitations but is portable and provides a good framework for experimentation. The second prototype draws upon the lessons learned from the first one but application isolation is provided through a modified runtime. Runtime changes lead to a robust and fully functional environment in which applications can be terminated at any instant and their resources can be accurately controlled.

Isolating applications from one another enables secure multitasking in the Java™ platform. However, there are other important multitasking issues: application termination, inter-application communication mechanisms, dealing with native code, and resource control. Some of these topics are the focus of several projects [3,4,9,16,31] and are beyond the scope of this paper. In other words, the approach presented in this paper deals only with the application isolation at the object level. In this respect, it is general and applicable, not only to the Java programming language, but to other object-oriented languages with similar features (e.g. static, or class, fields) as well. In the case of "pure" implementations, the presented technique may be enough to turn a language into a multitasking environment. In the case of more "contaminated" systems, issues such as native code and thread termination also need to be addressed.

A note on the terminology is appropriate here. The term "application" is used in an under-defined but intuitively understood sense. This is sufficient for the purposes of this paper. It is important to stress, though, that the mechanism proposed here can be used to implement protection domains in a broader sense [16]. An application can be a component, a servlet, a bean, etc., and, in these general settings, an application or service can consist of a collection of cooperating domains.

The rest of the paper is structured as follows. The basic idea is described in Section 2. Section 3 contains a description of our bytecode-editing prototype. Possible optimizations are the topic of Section 4. Performance data gathered from that prototype are analyzed in Section 5. The second prototype, based on the KVM and runtime modifications, is described in Section 6. Section 7 discusses the impact of the proposed architecture on security. Related work is summarized in Section 8. A summary section concludes the paper.

2 THE BASIC APPROACH

This section gives a high-level overview of the proposed protection model in the JVM and the intuition behind it. A simple way of explaining the model is to first think of a straightforward approach to multitasking in the Java™ application environment: all applications share all classes. The essential observation at this

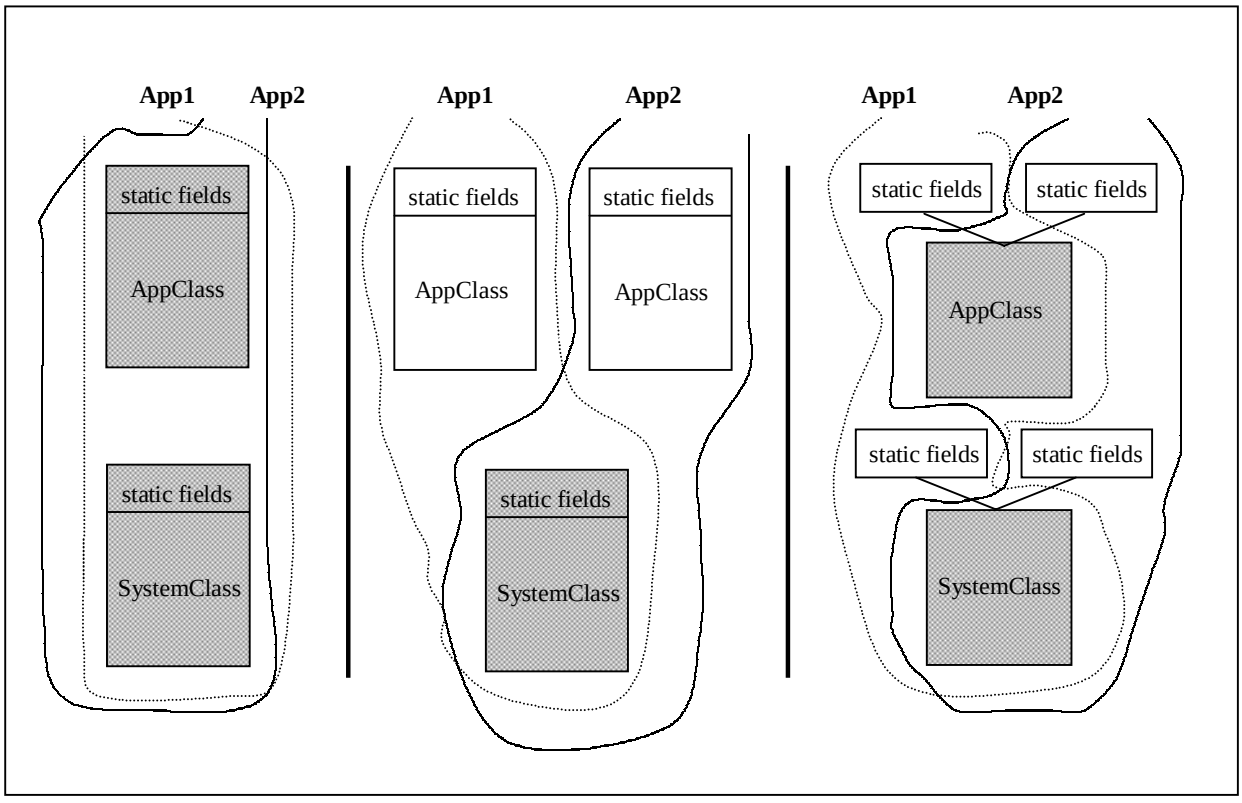

Figure 1. The naïve approach to Java multiprocessing ("share everything") is shown at the left; the center shows the class loader based solution; the right shows the new model. Shaded parts are shared by all applications; all other pieces are owned exclusively by a single application.

point is that a safe language already has some built-in support for isolating applications: data references cannot be forged, unsafe casting is not allowed, and jumping to an arbitrary code location is impossible. Consequently, the only data exchange mechanism (barring explicit inter-application communication) is through static fields. This can only occur either by explicit manipulation of static fields or by invoking methods which access these fields. It can lead to unexpected and incorrect behavior depending on how applications use the same class with static fields.

The above observation suggests an approach for achieving isolation among applications: to maintain a separate copy of the static fields for each class, one copy per application that uses the given class. However, only one copy of the *code* of any class should exist in the system, regardless of how many applications use it, since methods cannot transfer data from one application to another once the static fields communication channel is removed. (Dealing with covert communication channels is beyond the scope of this paper). Our proposal effectively gives each application the illusion that it has exclusive access to static fields while in reality each application has a separate copy of these fields.

In class loader based isolation, each application has a separate copy of the application classes (with static fields) but all system classes are shared. Two observations are important in this case. First, typically class loaders *do not share enough:* they replicate the code of application classes. Second, class loaders *share too much:* they share static fields of system classes. Addressing these two issues leads to the same model as above: all classes are shared but there is a per-application copy of each static field. Figure 1 contrasts the sharing and isolation in the most straightforward but simplistic approach, class loader based isolation, and the proposed model.

Our model combines the best features of the process and the class loader based approaches. First, many applications can execute in a single JVM. This has all the advantages of class loaders over processes: (i) switching from one application to another does not require a costly process context switch, (ii) startup time is faster, and (iii) the applications share the runtime resources, which improves the overall system scalability. Second, only one copy of a class is loaded into the system, regardless of how many applications use it. This improves over both existing approaches in terms of saved code space and saved repeated JIT compilation time. Third, applications are isolated from one another – they cannot exchange data through shared variables of any class. This is a vast improvement over what class loaders can offer. Finally, no new programming convention is introduced. In particular, the bytecode of existing applications does not have to be modified.

3 DETAILS

There are a number of details that need to be addressed to turn the idea presented above into a workable system. This section discusses a hypothetical source-to-source transformation implementation. Our first prototype, the performance of which is described in Section 5, performs these transformations but at the bytecode level. Bytecode-to-bytecode transformation is preferable over source-to-source transformation because often the source is not available. The source-to-source level is most appropriate for explaining the approach, though. Section 6 discusses functionally equivalent runtime modifications.

The general idea is to extract all static fields from each class and to transparently create a separate copy of these fields for each application. Consider a class X, containing static fields. It is split into three classes: the original one but without the static fields, a new class X$sFields containing all the static fields (which are now instance fields in X$sFields), and a new class X$aMethods. X$aMethods maintains an instance of X$sFields for each application; the methods of X$aMethods access the correct instance of X$sFields based on the application id extracted from the current thread. Only one copy of modified X and X$aMethods is present in the JVM regardless of the number of applications using the original class.

As an example, consider the following class:

```
class Counter {
  static int cnt;

  static {
    cnt = 7;
  }

  static void add(int val) {
    cnt = cnt + val;
  }
}
```

The transformations produce the three classes listed in Figure 2. The transformations affect only static fields and the way they are accessed. The first new class, Counter$sFields, contains all the static fields of Counter. The modifiers static, final, private, protected and public are removed from the fields so that they have package access. Thus, all static fields of Counter become instance, non-final, package-access fields of the new class Counter$sFields.

The second generated class is Counter$aMethods. It contains a table mapping application identifiers onto per-application copies of Counter$sFields. For each field from Counter$sFields there is a pair of get$() and put$() methods in Counter$aMethods. In our particular case there is only one static field and thus Counter$aMethods has two such access methods: put$cnt() and get$cnt(). Each of them looks up the copy of Counter$sFields corresponding to the current application and then accesses the named field. If the lookup does not succeed it means that this application's copy of Counter$sFields has not been generated yet and that the appropriate initialization has to be taken care of.

Let us note here that the field sfArr and the methods of Counter$aMethods could be stored in the original class file of Counter. This is possible for concrete classes only; interfaces cannot have non-abstract methods. In our base transformation, all class files are treated uniformly, though, and both the proper classes and interfaces are correctly taken care of.

The original class, Counter, undergoes the following modifications. All static fields are removed from Counter. A new method, hidden$initializer(), is added. It contains a modified code of the static initializer of Counter. It is invoked whenever a new application uses static fields of Counter for the first time.

Once these modifications are performed, the code of each has to be inspected (either off-line or at load time). Each access to a static field has to be replaced with the appropriate get$() or put$() method. At the bytecode-to-bytecode transformation level this becomes a replacement of each getstatic or putstatic with appropriate get$() or put$(), respectively.

```
class Counter$sFields {
  int cnt;
}


class Counter$aMethods {

  static Counter$sFields[] sfArr =
    new Counter$sFields[MAX_APPS];

  static Counter$sFields getSFields(){
    int id = Thread.currentAppId();
    Counter$sFields sFields;
    synchronized (Counter$aMethods.class) {
      sFields = sfArr[id];
      if (sFields == null) {
        sFields = new Counter$sFields();
        sfArr[id] = sFields;
        Counter.hidden$initializer();
      }
    }
    return sFields;
  }

  static int get$cnt() {
    return getSFields().cnt;
  }

  static void put$cnt(int val) {
    getSFields().cnt = val;
  }
}


class Counter {

  static void hidden$initializer() {
    Counter$aMethods.put$cnt(7);
  }

  static { hidden$initializer(); }

  static void add(int val) {
    int tmp = Counter$aMethods.get$cnt();
    tmp += val;
    Counter$aMethods.put$cnt(tmp);
```

Figure 2. Three classes generated from the example class Counter.

3.1 Thread Safety and Performance Considerations

The code presented in Figure 2 synchronizes on every static field access. Depending on the implementation of monitors, this can be expensive. Because of a subtlety in the memory model of the JVM [23], using the double-check idiom:

```
sFields = sfArr[id];
if (sFields == null) {
  synchronized(Counter$aMethods.class) {
    if (sFields == null) {
      sFields = new Counter$sFields();
      sfArr[id] = sFields;
      Counter.hidden$initializer();
    }
  }
}
```

in order to limit synchronization to only when sFields is actually constructed is not guaranteed to work. For details, see the discussion of double check and lazy instantiation idioms and of problems with volatile in [24]. In short, the problem boils down to the fact that memory updates performed in a synchronized section are not guaranteed to appear in their syntactic order to other threads. Whether this problem manifests itself or not depends on the particular JVM implementation. Interested readers are referred to [23].

Our first prototype, based on bytecode editing, has an option determining whether classes should be rewritten according to the pattern presented in Figure 2 or using the optimized code, presented above. Implementing the proposed isolation mechanism directly in the JVM may avoid entirely excessive synchronization.

3.2 Static Synchronized Methods

Suppose that add() of Counter is a synchronized method. This may lead to the following problem in the transformed code: one application calls add() and while the calling thread executes the body of the method, it is suspended by another thread from the same application. This is a serious denial-of-service problem since the suspended thread still holds a monitor and no other application is able to execute Counter.add(). Another scenario leading to the "capture" of a class monitor is an infinite loop in a static synchronized method. This problem does not exist if applications using the class Counter are loaded by separate class loaders.

These examples demonstrate that in the new isolation model, there is a need to have a separate monitor for each application. The objective is to ensure that proper mutual exclusion takes place among the threads of each application but one application cannot prevent another from using a given static synchronized method.

A relatively simple transformation, performed prior to the one shown in Figure 2, handles this problem. Implicit synchronization for static methods is replaced by explicit synchronization on the corresponding $sFields object owned by the current application. For example,

```
static synchronized void add(int val) {
  cnt = cnt + val;
}
```

is replaced by

```
static void add(int val) {
  Object sync;
  sync = Counter$aMethods.getSFields();
  synchronized(sync) {
    cnt = cnt + val;
  }
}
```

before it is subject to the static field replication.

3.3 Correctness and Security Issues

Since the transformations remove final and access control field modifiers when moving static fields into $sFields classes, it is important to ensure that the original program's semantics are preserved under the transformations. To this end, let us note that during the original compilation process, before the transformations, the source program has no knowledge of the $sFields and $aMethods classes and direct references to them will cause compilation errors. Reflection and handcrafted bytecode are the only ways via which the property of, for instance, being final or private can be violated. It is quite easy, though, to include appropriate checks in the implementation of reflective methods so that they cannot access generated classes. The reflection system can even be modified so that it behaves exactly as it did with the unmodified classes. Similarly, load-time checks can quickly detect and reject code that attempts to directly access the generated classes.

It is also important to make sure that the transformations preserve the property that a class is initialized before using of any of its instances. This is accomplished by inserting into the constructor code that makes sure the corresponding static initializer has been invoked. The code is similar to the getSFields() method in Figure 2. The double check optimization (Section 3.1) can be optionally applied.

3.4 Compatibility with Class Loader Based Multitasking

It is important to ensure that that static initializers are invoked in the same order and the same number of times as in class loader based multitasking. Compatibility reasons are not the only ones. For instance, static initializers of application classes may open network connections or load new (native) libraries. This may happen without accessing any static fields. This simple class below illustrates the problem:

```
class Test {
  static {
    System.out.println("Here");
  }
  static void dummyMethod() {}
}
```

Suppose two applications call Test.dummyMethod(). In a class loader based system, "Here" would be printed out twice. When executed in the isolation model proposed so far, the program would print the string only once.

To address this issue our current prototype invokes a static initializer of a class C when (i) the class C is loaded for the first time, and when another (i.e. not the one that caused (i)) application either (ii) accesses a static field of C for the first time or when it (iii) invokes a static method on C for the first time. The code for (ii) is shown in Figure 2; (iii) is accomplished by a flag check.

Another important property required by the language specification is that all superclasses are initialized before a class is initialized. This is accomplished by making sure that hidden$initializer() methods check whether the superclass has been initialized.

3.5 Atypical Cases

In several cases, the automatic transformations described above have to be augmented with manual re-coding of classes. Let us consider System.out. In most implementations of the JVM, this field is initialized by the runtime. It is important to ensure that each application has an access to the actual System.out (if a security policy of a given environment allows this) and that, at the same time, this static field is not directly shared by the applications.

In general, resources that must be shared by all classes have to be identified for each particular JVM. Manually dealing with them seems to be necessary for a handful of system classes only and wrapping selected classes with multiplexing/de-multiplexing code may be the most effective solution. For instance, in this prototype, each application sees a separate copy of System.out, but internally this print stream is implemented as prefixing each new line with the id of its application.

3.6 Shortcomings of Bytecode Editing

The prototype presented in this section treats the JVM as a "black box" and performs all the described actions at the bytecode level. This makes the prototype portable, but causes several problems as well.

Some issues are related to sharing of certain data structures by the runtime. For instance, typically there is a single internal string table. This can lead to unexpected sharing. Another instance of problematic sharing is finalization. In fact, finalizers are the only place we are aware of where the semantics of a program is not preserved by the bytecode-editing prototype. This is the case when finalization code uses static variables. Devising a general solution without modifying the runtime and without adding a field to each object seems to be difficult.

Finalizers that never terminate (e.g. because of an infinite loop) are another problem. Such behavior would capture the finalizer thread and would not allow it to finalize objects of other applications. Again, runtime modifications (e.g. separate per-application finalizers) seem to be necessary to address this problem in its generality.

Native code leads to another set of difficult issues, which can not be dealt with via bytecode editing. First, native methods may preserve state between invocations. This state may have to be replicated on a per-application basis in order to avoid clashes among applications. Second, application-defined native code has to be either well behaved, disallowed, or contained (e.g. in a separate process or via binary rewriting [31].) Everything executes in the same address space and unconstrained native code has unconstrained access to all data of other applications.

Another topic in this category is resource control. Isolation alone does not prevent denial-of-service attacks. To some extent, resources can be controlled through bytecode editing [9], but this

is not a complete approach. For instance, neither CPU usage nor memory allocated internally by some core classes can be controlled via bytecode rewriting only.

Our general approach does not preclude finalization or string interning. Neither does it imply giving up on user-defined native code or on resource control capabilities. The goal of the bytecode-editing experimental prototype is to explore some of the issues related to multitasking in the Java platform. In our opinion, a robust implementation addressing all of the problems mentioned in this subsection is possible, but requires runtime modifications.

4 OPTIMIZATIONS

The basic scheme described above has one very useful property: classes are modified one-by-one - there is no need to analyze another class before ending the modifications to the current class. Another property is portability. In this section, several simple optimizations are presented. They can be performed as source-to-source transformations. As such, they do not break portability but some may require analyzing more than one class before modifications to a particular class can be completed.

An important category of optimizations is preserving selected static final fields in their original classes. In such cases original getstatic (and, in initialization code, putstatic) instructions are left unmodified whenever accessing such preserved fields. This avoids having to look up the current application identifier and then to find the corresponding $sFields object.

Static final fields of primitive types can be preserved in their original classes since this cannot lead to any inter-application interference. The optimization makes it necessary to scan the bytecode of referenced classes in order to decide whether a field named in getstatic/putstatic is final or not.

Preserving static final strings in their original classes seems like another good candidate for an optimization. Strings are immutable so their fields or methods cannot act as a data communication channel between applications. However, if an application uses a static final string as a monitor object for a synchronized statement, another application may compete for the same monitor. Thus, preserving static final strings may lead to unwanted interference at the level of accessing mutual exclusion code.

Objects can be preserved in their original classes only if they are not used as synchronization objects and if they are immutable. Arrays of primitive types are a particular example. As will be shown later, leaving such immutable arrays in the original class may lead to significant performance gains. However, detecting array immutability (after its initialization in a static initializer) is impossible in general.

It is also important to point out that static final fields (both of primitive types and objects) can be preserved in their original classes only if they are initialized by constant values. This is a very typical case for primitive types and strings and is easy to detect. To see why this is important, let us consider this simple program:

```
public class C {
  final static String startTime =
    new java.util.Date().toString();

  public static void main(String[] args)
  {
    System.out.println(startTime);
                ...
  }
}
```

Preserving startTime in its original class (and, in effect, not replicating this field) would lead to the following behavior: *any* invocation of this program would print out the date of when the *first* copy of **C** was executed. This is different from the intended behavior to print out the starting time of the current instance of the program. Although programs are typically not written this way, our optimizations must not change the behavior of any program and thus the requirement that only constant-initialized static final fields can be preserved in their original classes.

Some further optimizations are possible. For instance, for actual classes (i.e. not interfaces) the new get$() and put$() methods can be added to the original class itself. This effectively merges the $aMethods classes into original classes. Initial experiments with this optimization did not indicate any performance gains or significant space savings and was not pursued further.

5 PERFORMANCE ISSUES

We implemented a prototype system based on bytecode editing. Bytecode is modified according to the transformations described in Section 3. The modifications can take place either off-line or at load time. The resulting bytecode is successfully verified by the verifier [21]. Since the prototype does not require any runtime changes, it effectively is a portable and transparent multitasking layer for the Java programming language. Portability of such a layer may be very valuable. As discussed below, our experience with the prototype is positive and the lessons learned easily translate into design decisions guiding runtime customization (Section 6.)

The experimental setup consists of a single processor UltraSPARC™ 167MHz, with 400 MB of RAM, running the Solaris™ Operating Environment, version 2.6. The Sun Microsystems Laboratories' Virtual Machine for Research[1] (ResearchVM) with 256MB of heap was used. Bytecode rewriting took place off-line (readers interested in rewriting classes on-line are referred to [9] for performance measurements of the bytecode editor used.) For the performance discussion, we have chosen six benchmarks from the JVM98 SPEC suite v. 1.03 [27]: db is a database application; jack is a parser generator; javac is the jdk1.0.2 compiler; jess is an expert system; mpegaudio is an audio file decompression tool; and raytrace is a raytracer (currently the JVM98 SPEC includes mtrt, which is a modified version of raytrace.)

Because of the VM's bootstrapping structure and complex interactions between the runtime and some classes, about two dozen system classes were not considered for rewriting (System, Thread, etc.) This small departure from rewriting all system classes can be avoided through a careful design of the bootstrapping process. It does not have any impact on the reported performance numbers.

[1] The same VM is embedded in Sun's Java 2 SDK Production Release for the Solaris Operating Environment, and is available at http://www.sun.com/solaris/java.

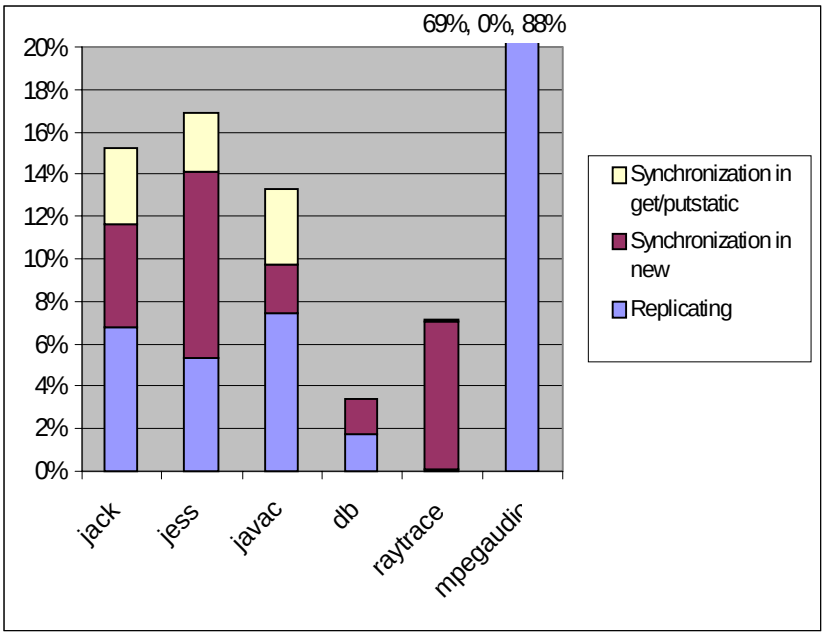

Figure 3. Performance overheads.

5.1 Time Overheads

In addition to running the benchmarks transformed with the basic modifications, several optimizations were implemented. First, static final fields of primitive types were preserved. Then, in addition, static final strings were preserved. Finally, immutable arrays of primitive types were preserved.

Figure 3 shows the overhead of running a single copy of a benchmark measured against the execution time of the unmodified program. The overheads for jack, jess, javac, db, and raytrace are less than 18% in the basic version of transformations; for mpeg, the overheads are 157%. Figure 3 separates out the costs of synchronization of each static field access (Section 3.1) and new object creation (Section 3.3). If the double check idiom (Section 3.1) is used, these synchronization overheads disappear. Figure 4 uses the experimental results of running the benchmarks with the double check idiom enabled as a starting point and then successively shows the improvements resulting from applying optimizations discussed in Section 4.

Static final fields of primitive types are a relatively frequent construct. In addition to eliminating additional access code, preserving them in their original classes allows the JIT to perform constant propagation. Preserving these fields brings about the biggest gains for these five benchmarks. Preserving static final strings has impact on jack and javac, which use many java.lang.String objects. For benchmarks other than mpegaudio, the first two optimizations reduce the overheads to between 0.2% (db, raytrace) to about 5% (javac, jess).

The mpegaudio benchmark, subject to our basic, unoptimized transformation, performs much worse than any other program we used to test our system. The reason is that mpegaudio relies heavily on static final arrays of floating point numbers. Bytecode inspection reveals that some of them are immutable. Preserving them in their original classes and accessing them via the unmodified getstatic instructions leads to a 24-fold overhead reduction – from 69.3% to 2.9%.

All the overheads include the JVM startup time and are reported as measured against the unmodified benchmarks. In particular, the benchmarks use the implicit system class loader. If it is replaced by an explicit, user-defined one, the performance of the original benchmarks worsens. When measured against such a lower reference point, the relative overheads reported for our model are lower by 15%, on the average, when compared to the overheads in Figure 4. The reason is that the necessary bookkeeping and file reading is slower through Java™ class libraries (java.io classes) than when performed by well-tuned native code.

Statistical information on executed bytecode instructions helps to interpret overhead data. The selected execution time statistics are contained in Table 1. When analyzed together with Figure 4, this

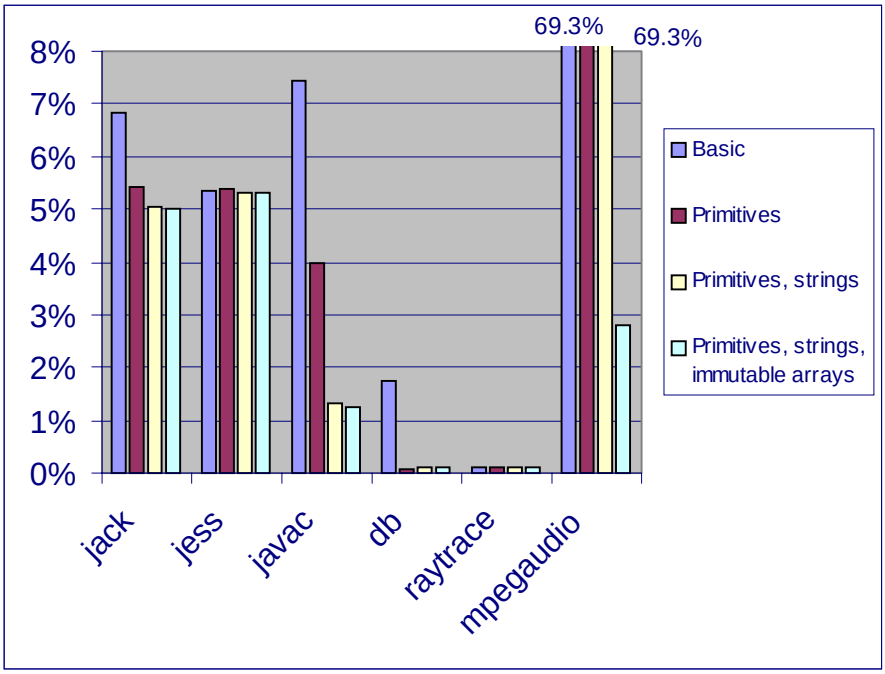

Figure 4. Performance overheads of the basic model and several optimizations.

	% of getstatic/putstatic bytecodes			
Benchmark	Basic transformation	Preserving primitive types	Preserving primitive types and strings	Preserving primitive types, strings and immutable arrays
jack	0.1483	0.1293	0.1146	0.1146
jess	0.1313	0.1313	0.1294	0.1294
javac	0.1228	0.0630	0.0235	0.0235
db	0.0047	0.0004	0.0004	0.0004
raytrace	0.0019	0.0018	0.0018	0.0018
mpegaudio	1.3547	1.3547	1.3547	0.0001

Table 1. Selected dynamic execution profiles.

data confirms the expected result: the runtime overheads are proportional to the number of getstatic and putstatic instructions replaced in the original programs.

In particular, the "*basic transformation*" column reports the total number of getstatic and putstatic instructions as the percentage of the total number of executed bytecodes. All these instructions are replaced by the basic transformation (Figure 2). On the average, 22 bytecodes (out of which three are method invocations) replace each getstatic and each putstatic. The last three columns in Table 1 contain the percentage of getstatic and putstatic instructions still emulated by the substituted replicating code under the optimizations (i.e. referring to fields not preserved by the optimizations).

5.2 Scalability

The data presented so far show the overheads of running a single application in our isolation model. Another insight into the performance of the new model is to run multiple copies of the same application.

5.2.1 Comparison with Class Loaders

First, the benchmarks were executed under the proposed isolation scheme and then under the class loaders control. Figure 5 shows typical results of such experiments. For reporting, we have chosen raytrace and jess – the benchmarks with the lowest and highest overheads, respectively, after the optimizations are applied. The total execution times of running multiple copies of the benchmarks are normalized to the execution time of a single copy of a benchmark loaded by a custom class loader. All implemented optimizations as well as the double check idiom are applied in the new model executions. Other benchmarks show very similar scalability behavior, but not all of them can be run with a substantial number of copies while using class loaders. For instance, only three copies of the jack benchmark can be run concurrently using class loaders; an attempt to run the fourth one is aborted because of the lack of space for the JITed code in the standard configuration of the chosen JVM. For raytrace, an attempt to execute 81 copies triggers the same abort message; for jess, it was possible to run only 18 copies.

For raytrace, the performance gap shows that the new model, initially about 4% slower than class loaders, is about 10% faster when 80 applications are executed simultaneously. This is not a surprising result and shows how much time is saved when no repeated compilations of the same classes are performed. The jess benchmark behaves similarly.

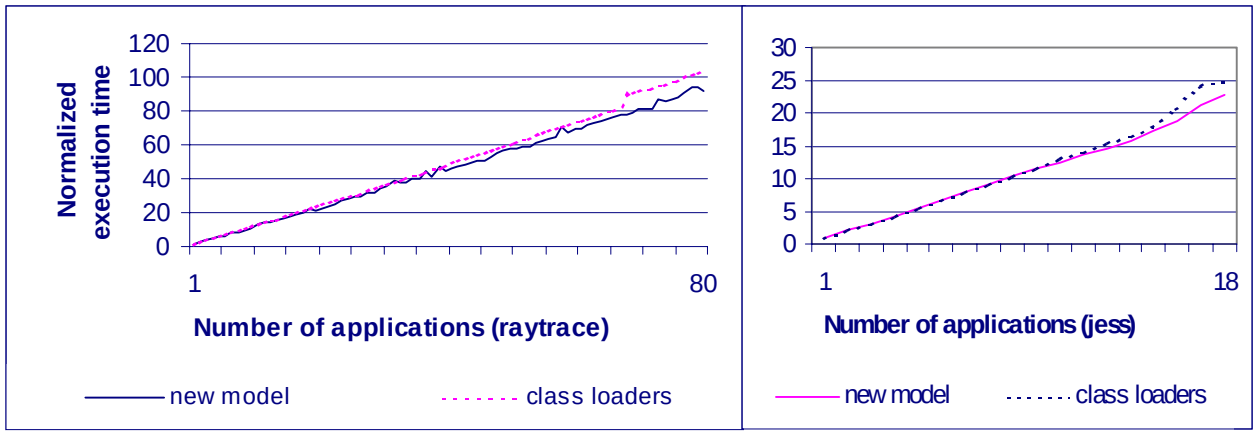

Figure 5. Performance of multiple simultaneous executions of raytrace and jess with isolation provided by the new model (solid line) and by class loaders (dotted line). The numbers are normalized to the single instance of an application loaded by a custom class loader. Lower is better.

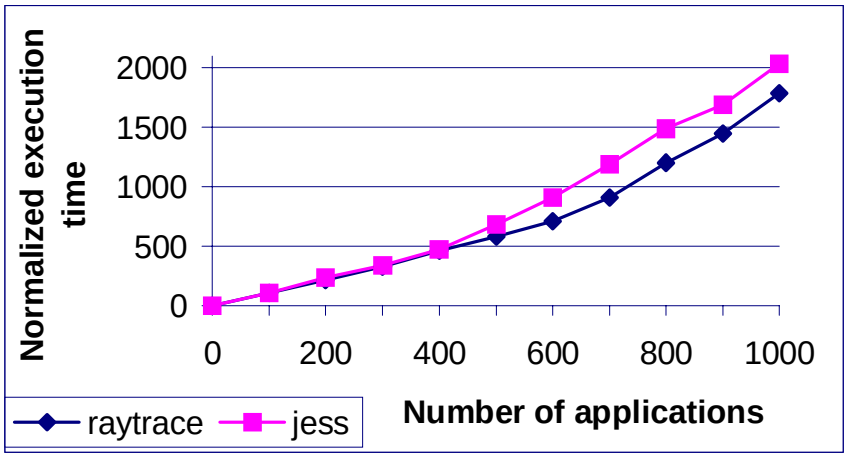

Figure 6. Total execution time of many copies of raytrace and jess.

5.2.2 Running Many Applications

The proposed model can actually support concurrent execution of many more applications than is possible with class loaders. For instance, we were able to run a thousand copies of raytrace. This was also possible for jess, but the benchmark had to be modified slightly in order to avoid having more files open that the underlying OS allows for a single process.

Figure 6 shows the total execution time. Each plot is normalized to the execution time of one copy of a benchmark. The upward curving means that the time to complete an instance of a benchmark increases with the number of applications executing now in the system. This is not surprising – for instance, managing threads becomes more expensive as their number grows.

5.2.3 New Model vs. Process-Based Model

The process-based approach scales poorer than class loaders based approach. Typically, running more than one JVM, each executing a single copy of a benchmark, results in about 10-15% overhead when compared to the new model. This may be acceptable in many cases. However, the main problem with the scalability of the process-based approach is that the maximal number of concurrently executing JVMs/applications is either small. For instance, on our software and hardware configuration at most up to 30 JVMs can run concurrently, each of them executing the same benchmark, for each benchmark.

5.3 Space Overheads

In order to estimate the space overhead issues, all classes from the JVM98 SPEC benchmarks were statically analyzed. 14.1% of all classes have static fields. 5.7% of all classes have static non-final fields. The first number indicates how many $aMethods and $sFields classes are typically generated from a JVM98 SPEC program in the absence of any optimizations. The second number is an estimate of how many such classes will be generated with all possible optimizations applied.

The size of an average $sFields generated from an application class is about 4.9 fields with no optimizations and 3.1 fields if all final fields can be preserved. For system classes these numbers can be higher. Code statistics obtained from the JDK1.2 classes show that 28.8% of classes have static fields; the average number of static fields per class that has them is 5.7. Similarly, 10.2% of all classes have static non-final fields and the average number of such fields is 2.7.

On the average, the static frequency of getstatic is 0.77% (about half of this on non-final fields); for putstatic this number is 0.15%. These numbers suggest how many method code transformations should be expected. Very similar frequencies can be found in the code of system classes.

5.4 Synchronized Static Methods

The time overhead introduced by the synchronized static method transformation is insignificant. Only ten bytecode instructions are added to each such method. The associated code expansion is also negligible since only 1.6% of all JVM98 SPEC classes have static synchronized methods and each such class has, on the average, only two such methods.

Even better news (statistically) is the analysis of the frequency of classes with static synchronized methods but without any static fields. Field-less $sFields have to be generated for such classes to replicate monitors. Only about 0.5% of classes from the JDK 1.2 qualify for this transformation. There is no such class in the whole JVM98 SPEC suite. Overall, this transformation does not lead to noticeable space or time overheads.

6 IN-RUNTIME IMPLEMENTATION

The overheads introduced by the portable prototype described above are on the order of ten per cent when measured on a high-performance JVM. Implementing the replication of static fields and monitors in the runtime instead of doing it at the bytecode level can improve performance. We did not go through this exercise for the ResearchVM for two reasons. First, we were content with the current overheads and removing the few percent did not justify a significant implementation effort, involving, among others, the JIT. We expect that putting the isolation into a high-performance JVM runtime can reduce the current overheads by an order of magnitude, down to a fraction of one per cent. This belief is based on our experiences with JRes [9], where performance overheads of memory accounting were an order of magnitude lower in an in-runtime prototype than for bytecode-editing.

Instead, we focused on providing multitasking for the KVM for the 3COM Palm Connected Organizer. Running concurrent multiple copies of the KVM was not feasible on the Palm. This combined with the lack of class loaders made our approach the only approach at all that enables multitasking in the Java programming language on the Palm. However, the low computing power of the device (2.7 MIPS at 16.6MHz processor clock) and the size of the current KVM heap (64KB) did not allow our rather complex bytecode editor to be run on the KVM/Palm. Running it off-line was also not desirable since it introduces additional software to be installed on a PC and the KVM would no longer be self-contained. The remaining option was to implement the isolation in the runtime.

The runtime modifications operate as follows. At load time, each class is examined and each static field is replicated n times (n is the maximum allowed number of applications). Similarly, class monitors for classes with static synchronized methods are replicated. No optimizations are performed for two reasons. First, on such a slow device they would significantly add to an application startup time. Second, the optimizations are less important when the runtime is modified. This is because getstatic and putstatic are implemented as well-tuned C code. Fetching an application id and using it to index an array of copies of a static field translates into less than ten machine instructions and no procedure calls. This compares with twenty-two bytecode instructions (three of which are method invocations) in the bytecode-editing prototype. This explains why there is virtually no performance degradation in this prototype. However, on a high performance JVM, where the overhead caused by our modifications are not dwarfed by the general slowness of the runtime (the KVM currently has no JIT, for instance) we expect the performance penalties to be less than one per cent.

When an application gets loaded into the modified KVM, it is assigned an application id or is rejected if no more application slots are available at the moment. Whenever a class is used by this application for the first time, the static initializer is run (when present), initializing the correct replicas of static fields.

It is important to stress a difference between the two prototypes we have built. The bytecode-editing system was constructed primarily as a portable reference implementation. It isolates applications, is easy to explain and modify in order to test out new concepts, and gives insight into sources of performance overheads. However, it tackles *only* isolation at the object and method level. As such, it is a useful research vehicle but impractical for some realistic applications, since it does not address native code issues, application termination, resource control, and inter-application communication.

The KVM-based prototype, on the other hand, is a fully functional multitasking JVM. In particular, applications can be stopped and killed without affecting the others, a copy-based inter-application communication interface is provided, and a JRes-like resource control API is present. The relative simplicity of the threading model (e.g. context switch can never happen before in the middle of the execution of a bytecode instruction) of the current version of the KVM proved to be very helpful during the implementation. However, the crucial enabling piece of technology was the isolation idea presented in this paper.

7 DISCUSSION

As has been argued, applications can be protected from one another both at the level of data access and at the level of access to static synchronized methods. The implications of the proposed architecture reach beyond providing inexpensive isolation. This section discusses these issues.

7.1 Subsuming the Role of Class Loaders

Class loaders provide multiple name spaces – two classes loaded by different class loaders are treated as distinct types even if they have the same name (and even if they come from the same file.) The goal is application isolation. The proposed new model achieves this isolation without incorporating the loader of the class into the type system. Since class loaders are not needed for separating name spaces anymore, it is natural to question other aspects of their usefulness in a system where isolation is provided by our new model.

Liang and Bracha [20] describe in detail the benefits of having class loaders in the JVM: lazy loading, type-safe linkage, user-definable class loading policy, and multiple name spaces. Lazy loading happens in our model as well but only once for each class. The linkage is as type safe in our model as when class loaders are used. Whenever a class loader would load a second copy of the same class, our model at most initializes corresponding copies of static fields of the class using its static initializer. The proposed model does not prevent users from defining policies concerning where from and how to fetch class files.

A deficiency of our approach when compared with class loaders is that only one version of any given class can be loaded into the system. Thus, dealing with changing class implementations and dynamically loading new versions of them for new instances of applications is impossible in our current design. However, in many cases versioning is not as important as scalability and full separation. Moreover, although it is sometimes possible to bypass problems with class versioning using class loaders, in general this issue cannot be solved so simply [20]. In the past, class loaders were a source of subtle but dangerous security problems [20,25].

Let us consider a JVM without class loaders but with the proposed isolation model. An advantage would be to have a much simpler type system. Security implications, described below, do not seem to indicate that removing class loaders from the Java programming language should lead to security problems. Overall, reducing class loaders to what their name suggests – fetching classes – appears to be acceptable. Of course, the details require more work. Moreover, we currently consider the proposed model as a *replacement* for class loaders with respect to isolation and security; the implications of having both class loaders *and* the proposed isolation model in the same JVM require more investigation.

7.2 Impact of the Proposed Architecture on Security

Class loaders are extensively used in the JVM to detect whether a given method call has originated from a class loaded by a particular class loader (which may indicate that the code can not be trusted) or whether no such class-loader-loaded method is on the call stack (which means that all the methods are from trusted classes).

Removing class loaders from the type system and reducing their language security role does not mean that classes can no longer be tagged with their origin, though. Tags can still be present, but in our opinion, moving from code-based to thread-based security may both simplify security in the Java programming language and simplify its use. In our second prototype, each thread belongs to one application or is a system thread. The application identifier determines the set of actions a given thread can perform. Our initial experience with this security model for multitasking on the Java platform is encouraging. A more detailed discussion is beyond the scope of this paper. It must be stressed here, though, that backward compatibility with existing APIs is an important concern.

7.3 Communication

The proposed isolation model does not imply any particular communication mechanism for applications executing in the same instance of the JVM. For the KVM prototype, we have implemented a simple copy-only interface. Objects are passed between applications by deep copy, which prevents any sharing. While it can be argued that sharing of objects may be useful in certain situations (for instance, sharing large files), copying of data may be acceptable in many cases, since non-communicating or rarely communicating applications are common [28].

Sharing of object references would make our model less simple. Moreover, well-known and difficult application termination problems and resource accounting issues would have to be addressed. The experience of the J-Kernel project [16], where most of the implementation effort went into providing secure and controlled sharing, indicates further that copy-only communication mechanisms may be preferable over sharing from the code maintenance and development effort perspective.

8 RELATED WORK

Currently available multitasking solutions for the Java programming language can be categorized as (i) ones that use the OS process model for protecting applications from one another, (ii) ones that modify the JVM runtime to emulate OS processes, and (iii) ones utilizing an abstraction of class loaders. Related projects are discussed below in their corresponding categories.

8.1 Process-Based Protection

Typically, resorting to an OS for protection implies running multiple copies of the JVM, one per application. This is expensive in terms of virtual memory resources and application startup time and scales poorly. However, applications are totally isolated from one another (although they can communicate for instance via sockets, RMI, etc).

Improving scalability within the process-based framework is a goal of a recent project carried out at IBM [10]. Multiple JVMs execute in separate processes but they share a memory region to store data that can be reused across multiple JVMs. This shared memory region is used by all JVMs to load, link, verify, and compile classes. None of these has to be repeated for a class that has already been loaded by another JVM. In particular, loading, linking, verifying, and compiling of system classes happens only once, for the first JVM on a given computer. Reusing JVMs is another improvement proposed in [10]. The main idea is that whenever an application has terminated without leaving any residual resources behind (e.g., threads, open file descriptors, etc) its JVM does not terminate but cleans up its state and is ready to run another application.

Advocates of process-based application separation in the Java application environment point out that a failure of one process terminates only this particular application and may potentially affect other applications only through an absence of service. Common wisdom states that processes are more reliable than implementations of JVMs. This reasoning implies that executing multiple applications in a single copy of the JVM puts them at a risk of being abruptly terminated because another application triggers an action that would cause the whole JVM to go down. However, it does not necessarily have to be so. Processes still execute on top of an underlying operating system and no major operating system kernel is guaranteed to be bug-free. Ultimately, one trusts software, whether it is an OS or a runtime of a safe language. The reliability issues of the Java platform and of an OS kernel are essentially the same, although so far much more effort was put into testing and debugging of OS kernels than into these of JVMs. Moreover, programs written in a safe language have less potential for crashing because of software problems.

Our idea of separating static data out of classes has its operating system predecessor and analogue. For a long time [1] UNIX® programs have been divided into data and code segments. Code segments can be shared between applications but data segments are replicated for each application that executes the code. An example of a non-UNIX commercial operating system where similar design principles have been applied is Windows NT [26].

An example of a safe language approach to designing operating systems is SPIN [5], written almost entirely in a safe subset of Modula-3 [22]. Software protection protects the OS kernel from dynamic extensions. We share the view of the SPIN authors that *protection is a software issue* [6]. In fact, with a well-designed inter-application isolation in a safe language, there should be no need for hardware protection.

8.2 Runtime Modifications

A project at the University of Utah [3], resulted in two operating systems, demonstrating how a process model can be implemented in the JVM. The first system, GVM, is structured much like a monolithic kernel and focuses on complete resource isolation between processes and on comprehensive control over resources. A GVM process consists of a class loader-based name space, a heap, and a set of threads in that heap. Each process has its own heap and all processes can access a special shared heap. For every heap, GVM tracks all references leading to other heaps and all references pointing into it. This information is used to implement a form of distributed garbage collection. The CPU management in GVM combines CPU inheritance scheduling [11] with the hierarchy introduced by thread groups: thread groups within processes may hierarchically schedule the threads belonging to them.

The second system, Alta, closely models a micro-kernel model with nested processes, in which a parent process can manage all resources available to child processes. Memory management is supported explicitly, through a simple allocator-pays scheme. The garbage collector credits the owning process when an object is eventually reclaimed. Because Alta allows cross-process references, any existing objects are logically added into the parent memory. This makes the parent process responsible for making sure that cross-process references are not created if full memory reclamation is necessary upon process termination. Both GVM and Alta are implemented as considerable modifications to the JVM.

NOMADS [29] is a mobile agent system. A custom JVM runtime supports the mobility of executing programs resource control. Each agent executes in a separate virtual machine, but all virtual machines are running in the same process, which leads to the sharing of all of the virtual machine code.

The isolation idea presented in this paper is more lightweight than the above designs. In particular, in our basic design the heap is physically shared by applications (although logically it contains disjoint graphs of objects belonging to different applications.) However, when scaling up to large heaps, physically disjoint per-

application heaps may become a better choice, depending on the garbage collector design. More study is needed in this area.

8.3 Class Loader Based Protection

A simple example of multitasking that utilizes class loaders is Echidna [15]. It is a class library, which, with a reasonable degree of transparency, allows multiple applications to run inside a single JVM. Applications can cleanly dispose of important resources when they are killed. For example, when a process is killed all its windows are automatically removed.

A more complex example of a class loader based approach to application protection is the J-Kernel [16]. The J-Kernel adds *protection domains* to the Java programming language and makes a strong distinction between objects that can be shared between tasks, and objects that are confined to a single task. Each domain has its own class loader. The system, written as a portable Java library, provides mechanisms for clean domain termination (e.g. no memory allocated by the task is "left over" after it is terminated) and inter-application communication (performed via deep object copies of method arguments and return values).

JavaSeal [7] is a library defining a secure mobile agent kernel. It provides a set of abstractions for constructing agent applications. Class loaders are used to provide separate name spaces and agent isolation. The second use of class loaders is to perform additional class verification (for instance, classes with finalizers containing loops are disallowed). Communication is accomplished by object serialization.

Balfanz and Gong based their design of a multitasking JVM on class loaders [4]. Their goal was to explore the use of the security architecture to protect applications from each other. The proposed extensions enhance the standard JVM so that it can support multitasking. An important part of the work is the clear identification of several areas of the JDK that assume a single-application model.

9 SUMMARY

The ability to execute multiple applications written in the Java programming language safely and efficiently on the same computer is becoming increasingly important. The shortcomings of current solutions have been discussed in detail in the paper. Their analysis led to a new approach to multitasking on the Java platform. Design and technical issues have been presented. A portable bytecode-editing prototype has been used to prove the concept and to understand some performance issues.

This prototype serves as a good experimental platform to learn about design tradeoffs and performance issues, and can be used as a portable multitasking platform. The lessons learned from the implementation are important when adding our isolation mechanisms to the JVM runtime. Incorporating our approach in the runtime (i) reduces the overheads, (ii) simplifies the implementation, and (iii) removes bytecode editing from the critical "fetch class file-load-execute" path. In particular, the experience gained with this first prototype was very valuable during the design and implementation of the KVM-based prototype, where the isolation model is implemented entirely in the runtime.

Our isolation mechanism can be readily used on a whole range of computer systems. On large servers, one big JVM can contain many applications, which conserves system resources and helps scalability. On small systems, minimizing memory footprint is vital, and depending on the virtual machine and OS features no other multitasking alternative may be available.

For a pure and reliable Java programming language implementation, without features such as native code, the techniques presented in this paper lead to inter-application isolation equaling that of an operating system but without scalability penalties of process-based multitasking. For high-performance implementations with native code, JIT, and complex thread implementations, our techniques offer a core isolation framework but certain VM-specific issues have to be separately addressed to achieve the desired protection, resource control, and scalability levels.

10 ACKNOWLEDGEMENTS

The author is grateful to Ole Agesen, Malcolm Atkinson, Laurent Daynes, Mick Jordan, Doug Lea, Brain Lewis, Tim Lindholm, Bernd Mathiske, Ron Mann, Bill Pugh, Michael Shafae, Antero Taivalsaari, David Ungar, and Mario Wolczko for their help.

11 REFERENCES

[1] Arnold, J. *Shared Libraries on UNIX System V.* Summer USENIX Conference, Atlanta, GA, 1986.

[2] Arnold, K., and Gosling, J. *The Java Programming Language*. Second Edition. Addison-Wesley, 1998.

[3] Back, G, Tullmann, P, Stoller, L, Hsieh, W, and Lepreau, J. *Java Operating Systems: Design and Implementation.* TR UUCS-98-015, Department of Computer Science, University of Utah, August 1998.

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

[5] Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C.. *Extensibility, Safety and Performance in the SPIN Operating System*. 15th ACM Symposium on Operating Systems Principles, Copper Mountain, CO, December 1995.

[6] Bershad, B., Savage, S., Pardyak, P., Becker, D., Fiuczynski, M., Sirer, E. *Protection is a Software Issue*. 5th Workshop on Hot Topics in Operating Systems, Orcas Island, WA, May 1995.

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

[8] Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. *Compiling Java Just in Time.* IEEE Micro, May/June 1997.

[9] Czajkowski, G., and von Eicken, T. *JRes: A Resource Control Interface for Java.* In Proceedings of of ACM OOPSLA'98, Vancouver, BC, Canada, October 1998.

[10] 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.

[11] Ford, B. and Susarla, S. *CPU Inheritance Scheduling.* 2nd Symposium on Operating Systems Design and Implementation, Seattle, WA, October 1996.

[12] Gong, L. *Java Security: Present and Near Future*. IEEE Micro, 17(3), May/June 1997.

[13] Gong, L. and Schemers, R. *Implementing Protection Domains in the Java Development Kit 1.2.* Internet Society Symposium on Network and Distributed System Security, San Diego, CA, March 1998.

[14] Gosling, J., Joy, B., Steele, G. and Bracha, G *The Java language specification.* 2nd Edition. Addison-Wesley, 2000.

[15] Gorrie, L. *Echidna – a Free Multitask System in Java.* http://www.javagroup.org/echidna.

[16] 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.

[17] Sun Microsystems, Inc. *JavaBeans.* http://java.sun.com/beans/index.html.

[18] Sun Microsystems, Inc. *Java Servlet API.* http://java.sun.com/products/servlet.

[19] Sun Microsystems, Inc. *The K Virtual Machine – A White Paper*. http://java.sun.com/products/kvm/wp.

[20] Liang S., and Bracha, G. *Dynamic Class Loading in the Java Virtual Machine*. ACM OOPSLA'98, Vancouver, BC, Canada, October 1998.

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

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

[23] Pugh, W. *The Java Memory Model.* http://www.cs.umd.edu/~pugh/java/memoryModel.

[24] Pugh, W. *Fixing the Java Memory Model*. ACM Java Grande, San Francisco, CA, July 1999.

[25] Saraswat, V. *Java is not type-safe*. http://www.research.att.com/~vjbug.html.

[26] Solomon, D. *Inside Windows NT.* Second Edition. Microsoft Press, 1998.

[27] Standard Performance Evaluation Corporation. *SPEC Java virtual machine benchmark suite.* August 1998. http://www.spec.org/osg/jvm98.

[28] Spoonhower, D., Czajkowski, G., Chang, C-C., Hawblitzel, C., Hu, D., and von Eicken, T. *Design and Evaluation of a Extensible Web and Telephony Server based on the J-Kernel.* TR98-1715, Department of Computer Science, Cornell University.

[29] Suri, N., Bradshaw, J., Breedy, M., Groth, P., Hill, G., Jeffers, R., and Mitrovich, T. *An Overview of the NOMADS Mobile Agent System.* 2nd International Symposium on Agent Systems and Applications, ASA/MA2000, Zurich, Switzerland, September 2000.

[30] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. *Efficient Software Fault Isolation.* 14th ACM Symposium on Operating Systems Principles, Asheville, NC, December 1993.

[31] Wallach, D., Balfanz, D,. Dean, D., and Felten, E. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997.