CoDeSe: Fast Deserialization via Code Generation

Milos Gligoric Darko Marinov Sam Kamin
Deptartment of Computer Science
University of Illinois, Urbana, IL 61801, USA
{gliga, marinov, kamin}@illinois.edu

ABSTRACT

Many tools for automated testing, model checking, and debugging store and restore program states multiple times. Storing/restoring a program state is commonly done with *serialization/deserialization*. Traditionally, the format for stored states is based on *data*: serialization generates the data that encodes the state, and deserialization *interprets* this data to restore the state. We propose a new approach, called *CoDeSe*, where the format for stored states is based on *code*: serialization generates code whose execution restores the state, and deserialization simply *executes* the code. We implemented CoDeSe in Java and performed a number of experiments on deserialization of states. CoDeSe provides on average more than 6X speedup over the highly optimized deserialization from the standard Java library. Our new format also allows simple parallel deserialization that can provide additional speedup on top of the sequential CoDeSe but only for larger states.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*Code generation*

General Terms: Performance

Keywords: Deserialization, code generation

1. INTRODUCTION

Many tools for automated testing, model checking, and debugging explore program behavior by storing and restoring (parts of) program states multiple times [4, 6, 8, 9, 13, 16, 19, 20, 25–28, 31, 39, 41]. These tools exhibit two general patterns of restoring states, even when restoring is not the most expensive operation: (1) the same state (relatively small) is restored multiple times in the same program run, or (2) one state (relatively large) is restored only once during a run, but there are multiple such runs. In both patterns a state is stored once but restored potentially many times. The tools usually store and restore the state using *serialization* [21] (also known as *marshalling*) and *deserialization* (also known as *unmarshalling*), respectively. Serialization translates (a part of) the program state to a format that can be later used to restore the state with deserialization.

As an example in automated testing, consider OCAT [16]. OCAT improves the random generation of unit tests in Randoop [32] by capturing objects from actual executions, potentially mutating these objects, and using them to seed random generation. Randoop outputs unit tests as sequences of calls to the methods from the code under test, and OCAT adds calls to special methods that deserialize the captured objects. Hence, OCAT deserializes objects during both generation and execution of the unit tests. During generation, OCAT can deserialize an object a large number of times based on random selection. During execution, OCAT can deserialize an object many times if it appears in different tests (or several times in the same test).

As an example in program model checking, consider Java PathFinder (JPF) [15, 39]. JPF is a backtrackable Java Virtual Machine (JVM) that can systematically explore a given Java program by controlling non-deterministic choices due to thread scheduling and explicit non-deterministic calls. During exploration, JPF restores a program state several times to explore different executions that can lead from that state. (The number of times a state is restored during model checking varies greatly, with an average of 3, but the maximum ranging from 4 up to 200 [33].) While a program state can be restored by re-executing the program from the beginning [10, 30] or by undoing state changes for the depth-first search order [12], JPF for most search orders conceptually serializes and deserializes the entire JVM states. Each JVM state includes a stack (local per thread) and a heap.

As an example in debugging, consider efficient checkpointing by Xu et al. [41], which we will refer to as XRTQ. XRTQ restores the state of a long-running Java program without using a specialized JVM as in JPF (but XRTQ handles only single-threaded programs). Because deserializing the stack is hard without JVM support, XRTQ restores the state by combining re-execution of a simplified version of the program (to restore the stack) and deserialization of a part of the state (to restore the heap). Using this combination, XRTQ restores the state much faster than re-executing the full program. Note that XRTQ deserializes the state only once per a JVM run, but it is commonly a large state.

Serialization and deserialization are important not only in these testing tools but also in general applications to store and communicate parts of state. *Traditionally*, the format for stored states is based on *data*: serialization generates the data that encodes the state, and *deserialization interprets this data* to restore the state. The data encoding can be either binary, which is more compact, or textual, such as XML [23, 40], which is more readable and portable. Since serialization and deserialization are frequently used, the standard libraries for some languages, such as Java, provide highly optimized generic implementations of these operations. Moreover, techniques were proposed for speeding up these general operations, mostly serialization, by specializing through program generation [3, 24, 29, 38] or by incremental computation [1, 2]. However, the proposed techniques still use the same traditional format for stored states based on data.

This paper makes the following contributions:

Approach: We propose a new approach, called *CoDeSe* (COde-based DEserialization and SErialization), where the format for stored states is based on *code*: serialization generates code whose execution restores the state, and *deserialization simply executes this code*. Our goal for CoDeSe is to *make deserialization as fast as possible*. Effectively, we optimize for the scenarios where a state is restored potentially many times, while the traditional serialization and deserialization need to balance several requirements: efficient serialization, compact size of stored state, and efficient deserialization.

Implementation: We implemented CoDeSe as a Java library that provides an API similar to the standard Java serialization/deserialization library. The standard library has a number of features [21]: it provides default serialization of all non-`static` and non-`transient` fields for objects of classes marked with the `Serializable` interface and allows expressing specialized serialization in several ways, such as implementing private `writeObject`/`readObject` methods or the `Externalizable` interface. Our CoDeSe library fully supports the default serialization/deserialization and commonly used specialized features from the standard library but does not support some features such as code evolution (Section 3.8).

Evaluation: We performed a number of experiments to evaluate deserialization time for states captured during the execution of real applications, including some states directly taken from the OCAT experiments [16] and some scenarios similar to the XRTQ study [41], and for synthetic states from our running example. The experiments involve states of various sizes and patterns when a state is restored once or more times during one run. We compare the deserialization time using the standard Java library (SJL) and the CoDeSe library. CoDeSe is, on average, over 6X faster than SJL.

We also experimented with a simple parallel deserialization in CoDeSe. While the traditional deserialization interprets the serialized data sequentially to restore the state, our CoDeSe format allows multiple threads to execute deserialization code in two phases: creating objects and setting fields. The parallel CoDeSe can provide additional speedup on top of the sequential CoDeSe but only for larger states.

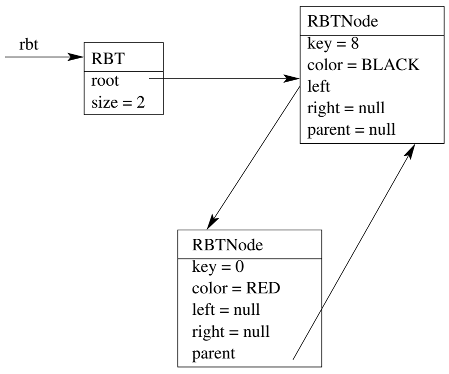
The experimental dataset that is used in the evaluation is publicly available at `http://mir.cs.illinois.edu/codese`.

2. EXAMPLE

To illustrate CoDeSe, we discuss serialization/deserialization of a state that consists of one red-black tree (RBT) data structure. We use RBT as our running example because it involves only two classes and simplifies the presentation. However, we point out that our experiments, which are described in Section 4, use states taken from real applications with many more classes.

```
class RBT implements Serializable {
    RBTNode root;
    int size;
    static boolean RED = true;
    static boolean BLACK = false;
    static class RBTNode implements Serializable {
        int key;
        boolean color;
        RBTNode left, right, parent;
    }
}
```

(a)



(b)

Figure 1: (a) Part of the `RBT` class; (b) An example RBT instance.

Figure 1(a) shows the relevant parts of the Java code that declares a RBT. The `RBT` class represents a tree. It has two instance fields: `root` is a reference to the root node, and `size` is the number of nodes in the tree. The `RBTNode` class represents a node. It has five instance fields: `key` stores the node value, `color` is `RED` or `BLACK`, `left` and `right` point to the children, and `parent` points to the parent.

Figure 1(b) shows an example instance of a RBT. The variable `rbt` points to the tree. Each box represents an object, showing the type and values of instance fields. There are three objects in the *object graph* reachable from `rbt`: one instance of `RBT` and two instances of `RBTNode`. We next describe serialization/deserialization of this RBT instance using the standard Java library and our CoDeSe library.

2.1 Background: Standard Java Library

We first show the API interface for serializing and deserializing the `rbt` object graph using the standard Java library (SJL). We then show the format for the stored object graph produced by the standard serialization. We finally discuss the algorithm used in the standard deserialization.

API: SJL supports serialization/deserialization through classes provided in the `java.io` package. It requires that the classes to be serialized and deserialized, such as `RBT` and `RBTNode`, be marked with the `java.io.Serializable` interface. (CoDeSe and some non-standard Java libraries, such as XStream [40] or JSX [23], do not require this interface; however, we do not compare CoDeSe performance with these other libraries because they are not tightly integrated with JVM and are thus often not as efficient as the SJL.) Serialization/deserialization is performed by writing/reading through the `ObjectOutputStream`/`ObjectInputStream` to/from the underlying stream (which can be, for example, a file on disk with `FileOutputStream`/`FileInputStream`).

```
ObjectOutputStream oos = new ObjectOutputStream(...);
oos.writeObject(rbt);
...
ObjectInputStream ois = new ObjectInputStream(...);
RBT o = (RBT) ois.readObject();
```

(a)

```
OBJECT_FLAG // (new object) rbt (mapped to handle 0)
RBT // rbt: class name
OBJECT_FLAG // (new object) rbt.root (mapped to handle 1)
RBTNode // rbt.root: class name
8 // rbt.root.key = 8
false // rbt.root.color = BLACK
OBJECT_FLAG // (new object) rbt.root.left (mapped to handle 2)
0 // rbt.root.left.key = 0
true // rbt.root.left.color = RED
NULL_FLAG // rbt.root.left.left = null
NULL_FLAG // rbt.root.left.right = null
REFERENCE_FLAG // (handle)
1 // rbt.root.left.parent = rbt.root
NULL_FLAG // rbt.root.right = null
NULL_FLAG // rbt.root.parent = null
2 // rbt.size = 2
```

(b)

```
1 // mapping: unique object handle to instance
2 Map<int, Object> handleToInstance;
3 Object readObject() {
4   switch (currentByte()) {
5   case OBJECT_FLAG:
6     ClassDescriptor desc = readClassDescriptor();
7     Object obj = desc.newInstance(); // create an object
8     int handle = nextHandle();
9     handleToInstance = handleToInstance ∪ {handle → obj};
10    // read fields
11    List<FieldDescriptor> fields = readFieldDescriptors(obj);
12    foreach (field in fields) {
13      if (field.isPrimitive())
14        switch(field.type()) {
15        case int: setIntField(obj, readInt());
16        case // ... other primitive types
17        }
18      else
19        setObjField(obj, readObject());
20    }
21    return obj;
22  case REFERENCE_FLAG:
23    int handle = readInt();
24    if (handle ∉ handleToInstance)
25      ... // complex code for deserializing cycles in graphs not shown
26    else return handleToInstance(handle);
27  case NULL_FLAG:
28    return null;
29  case // special cases, e.g., Class
30  }
31 }
```

(c)

Figure 2: (a) Example user code for serializing and deserializing in Java; (b) Serialization output for Java library; (c) Pseudo-code of deserialization from the `readObject` method in `java.io.ObjectInputStream`.

Figure 2(a) shows the user code to serialize the object graph reachable from `rbt` and deserialize it into another object graph `o`. For serialization, the code creates an instance of `ObjectOutputStream` and invokes on it the method `writeObject`, passing the reference to the root of the object graph to be serialized (`rbt`). For deserialization, the code creates an `ObjectInputStream` and invokes `readObject` on it that returns the reference to the root of the deserialized object graph.

Serialization Format: The `writeObject` method traverses the objects reachable from its given argument and outputs a *sequence of bytes* that encode values of (non-`transient`) instance fields of those objects. If a field is a reference, an object handle (a unique instance id) is written to the stream. Serialization needs to support aliasing of references and cycles in object graphs. Figure 2(b) shows the complete output of serialization for our example instance but shows abstract values rather than concrete bytes.

Deserialization: Figure 2(c) shows a simplified pseudo-code of the `readObject` method from the SJL. To restore the object graph, deserialization effectively *interprets the data* written by serialization. The key parts of deserialization are: (1) line 7 that creates objects for the new graph, (2) lines 12-20 that restore field values (including a recursive call to `readObject` for reference fields), and (3) lines 22-26 that provide appropriate reference values and support the complex case of cycles in the graphs, where some assignments need to be postponed until the objects are created.

```
CodeseOutputStream cos = new CodeseOutputStream(...);
cos.writeObject(rbt);
...
CodeseInputStream cis = new CodeseInputStream(...);
RBT o = (RBT) cis.readObject();
```

(a)

```
static Class clz0 = Class.forName("RBT");
static Class clz1 = Class.forName("RBT$RBTNode");
Object readObject() {
    Object[] objs = new Object[3];
    createObjects(objs);
    setFields(objs);
    return objs[0];
}
void createObjects(Object objs[]) {
    obj[0] = unsafe.allocateInstance(clz0); // rbt
    obj[1] = unsafe.allocateInstance(clz1); // rbt.root
    obj[2] = unsafe.allocateInstance(clz1); // rbt.root.left
}
void setFields(Object objs[]) {
    unsafe.putInt(obj[0], 8l, 2); // rbt.size
    unsafe.putObject(obj[0], 12l, obj[1]); // rbt.root
    unsafe.putInt(obj[1], 8l, 8); // rbt.root.key
    // unsafe.putBoolean(obj[1], 12l, false); // rbt.root.color
    // unsafe.putObject(obj[1], 16l, null); // rbt.root.parent
    unsafe.putObject(obj[1], 20l, obj[2]); // rbt.root.left
    // unsafe.putObject(obj[1], 24l, null); // rbt.root.right
    // unsafe.putInt(obj[2], 8l, 0); // rbt.root.left.key
    unsafe.putBoolean(obj[2], 12l, true); // rbt.root.left.color
    unsafe.putObject(obj[2], 16l, obj[1]); // rbt.root.left.parent
    // unsafe.putObject(obj[2], 20l, null); // rbt.root.left.left
    // unsafe.putObject(obj[2], 24l, null); // rbt.root.left.right
}
```

(b)

Figure 3: (a) Example API for serializing and deserializing in CoDeSe; (b) Serialization output for CoDeSe library.

2.2 CoDeSe Library

We first show how the API interface for serializing and deserializing the `rbt` object graph with CoDeSe is similar to SJL. We then show our novel format for the stored object graph produced by the CoDeSe library. The format is based on code and substantially differs from the traditional serialization format based on data, which is used by the SJL. Our format allows deserialization to simply execute the code.

API: Figure 3(a) shows the user code to serialize the object graph reachable from `rbt` and deserialize it into another object graph reachable from `o`. Compared to Figure 2(a), the only difference is using the two new classes `CodeseOutputStream/CodeseInputStream` that CoDeSe provides as subclasses of `ObjectOutputStream/ObjectInputStream`.

Serialization Format: CoDeSe introduces a new format for the serialized object graph such that deserialization need not interpret data but can directly execute code. More precisely, the output of CoDeSe serialization is the code whose execution restores the object graph. The process of serialization in CoDeSe is similar to the one from SJL: the `writeObject` method traverses the objects reachable from its given argument but outputs *code* rather than a sequence of bytes. CoDeSe serialization properly supports aliasing of references and cycles in object graphs.

Figure 3(b) shows the complete output of serialization for our example instance. The generated code uses the `sun.misc.Unsafe` class to create instances and set the fields. Note that SJL and some research projects, e.g. XRTQ [41], also use `Unsafe` during serialization/deserialization. The `allocateInstance` method allocates memory for an instance of a given type without running any of the constructors. The various `put` methods store a given value into the field at a given *offset*. For example, the offset for `RBT.size` was `8` on the JVM that we used. While the offset for a given field may differ between JVMs, it is constant for a fixed JVM (and often between JVMs). Our serialization uses the `Unsafe` class to obtain the offset and then hard codes this offset in the generated code.

The `readObject` method from Figure 3(b) can restore only the specific RBT object graph; in effect, this method is a specialized version of the generic `readObject` method from Figure 2(c) for Java. The specialized `readObject` method contains the key parts of deserialization: (1) it first creates all the objects for the new graph, (2) it then sets their field values including references that link the objects, and (3) it easily supports cycles in the graphs by executing `createObjects` and `setFields` in order. The example RBT instance has a cycle as `root.left.parent == root`.

Note that some lines in `setFields` are commented out. Those lines would set the fields to the default values for their types, e.g., `0` for integers, `false` for booleans, and `null` for references. CoDeSe format allows omitting those lines because creating objects in Java already sets all the fields to their default values. However, the standard serialization format includes even the default values because it cannot easily mark which object fields have such values.

While our running example has only three objects, the experiments in Section 4 consider objects graphs of various sizes and types, as well as various number of deserializations of the same graph in one JVM run. For example, for RBTs with 10 nodes CoDeSe deserialization is 2-4X faster than the standard Java deserialization, and for RBTs with 1000 nodes CoDeSe is 6-11X faster.

3. IMPLEMENTATION

While the high-level idea of CoDeSe is appealing—enable faster deserialization by generating code rather than data—providing an implementation that both offers efficient deserialization and handles states from real applications poses a number of technical challenges. We next describe our CoDeSe implementation.

3.1 Creating Objects and Setting Fields

Figure 3(b) shows the code that CoDeSe produces for the example instance. The code uses the `Unsafe` class that offers a low-level, efficient API for memory manipulation. An alternative would be to create objects using `new` and set values using the standard field derefencing, e.g., `rbt.size = 2`. However, such code may not compile because (1) the classes `RBT` and `RBTNode` may not have public default constructors (and even if there are constructors, they may have side benefits besides allocating objects), and (2) fields may not be publicly accessible (even if the code properly downcasted the objects from the `objs` array). Another, more expensive alternative would be to use Java reflection.

3.2 Multiple Methods and Classes

Figure 3(b) shows an output of CoDeSe serialization that has one method to create objects and one method to set field values. In general, each phase has to be split—object creation and field setting—into multiple methods because a method in Java is limited to 65536 bytes. There are other limits for Java classfiles [17], so CoDeSe may generate multiple classes for the code that deserializes one (large) object graph. CoDeSe appropriately adds hierarchical calls to all the methods from all the classes that it generates. While splitting each phase into multiple methods (or classes) is necessary for larger object graphs, such splitting could be beneficial even when not required by size, because it allows parallel execution of deserialization.

3.3 Parallel Deserialization

During deserialization, the object-creation phase has to finish before the field-setting phase starts to ensure that all objects are created before reference fields are set. While the phases have to follow sequentially, the work within a phase can be parallelized. Indeed, having code as the result of the CoDeSe serialization, and especially having multiple methods for each phase, allows the opportunity to deploy multiple threads to perform deserialization.[1] The maximum number of threads to be used in deserialization can be given as the argument to CoDeSe serialization such that it splits the work into enough methods even when not required by the limits on method size. Our implementation of CoDeSe serialization splits the work such that each method creates/sets approximately the same number of objects/fields.

Our implementation of CoDeSe deserialization uses the standard `java.util.Thread` objects for parallel execution and uses a `java.util.concurrent.CyclicBarrier` object for delineating the two phases. During each phase, each thread executes approximately the same number of methods, which provides good load balancing.

3.4 Default and Literal Values

Having code as the result of serialization offers other opportunities for optimizations. Recall from Section 2 that CoDeSe does not generate code to explicitly set the default field values. In contrast, the standard Java deserialization does unnecessarily set these values because it does not selectively avoid fields for *some* objects (although the private `writeObject`/`readObject` methods can avoid fields for *all* objects of some type).

CoDeSe creates boxed primitive values and `String` objects using literals and publicly available constructors (e.g., `new Integer(5)` or `new String("foo")`). We experimented with

[1]Note that parallelizing the traditional deserialization from one sequence of bytes would be much harder, although one can envision a new binary data format that encodes multiple sequences of bytes, and for XML-based serialization formats one can consider parallel parsing [37].

objects of these types and noticed significant performance improvement when treating these types as a special case instead of using the general `sun.misc.Unsafe` to create the objects and set their fields. CoDeSe still preserves the aliasing such that all the references that point to the same object in the original object graph point to the same (newly created) object in the restored object graph. Hence, it generates `new String("foo")` rather than just `"foo"`. In other words, CoDeSe restores an object graph isomorphic [14] to the original object graph.

3.5 Non-default writeObject/readObject

The `writeObject`/`readObject` methods provided on streams (as shown in figures 2(a) and 3(a)) by default serialize/deserialize all non-`static`, non-`transient` fields. SJL allows classes to implement their private `writeObject`/`readObject` methods that can select the fields to be written/read to/from the stream. For example, the class `java.util.TreeSet` has such private methods. This class implements a set using red-black trees. While for illustrative purposes our running example showed how to serialize an entire RBT object graph, note that serializing a set does not require storing the entire graph. Instead, one can only store and restore the elements in the set (and not the nodes that contain elements), which results in smaller stored data and restores the object at the abstract level (the set with the same elements) but not necessarily at the concrete level (the same RBT structure).

Our current CoDeSe implementation supports the private `writeObject`/`readObject` methods. If an object, say `objs[i]`, has such methods, CoDeSe serialization first collects the values written by `objs[i].writeObject(...)`, and then in the generated code, adds calls to `objs[i].readObject(...)` rather than setting the field values for `objs[i]`. The parameter that is given to the `readObject` call returns the values collected from the `writeObject` call, respecting the order in which the values were written. Our initial CoDeSe prototype did not support the private `writeObject`/`readObject` methods and thus incorrectly restored some objects that depend on hash values, which is an issue also discussed by XRTQ [41].

3.6 JNI

The default code generator in the CoDeSe library generates Java code that can be executed directly on any JVM (which supports `sun.misc.Unsafe` or a similar class). We also developed another code generator that generates native/C code that can be executed through the Java Native Interface (JNI). JNI [18] is an API that allows Java code to call to and be called from native code, for example written in C. Our JNI code generator produces the same set of Java classes as the default code generator, and these classes have the same set of methods, with one key difference: the methods have no Java body, but their declarations include the `native` modifier. For each Java class, our JNI code generator produces the corresponding native code that implements the methods from the class. These methods create objects and set field values through the standard JNI functions, such as `NewObject` and `SetObjectField`. While the Java code generated by the default code generator can easily be moved from one environment to another (e.g., an object graph is serialized on one computer and restored on another computer with the same JVM), the native code generated by the JNI code generator is not as portable because it is platform dependent.

3.7 Serialization

Our goal for CoDeSe is to *make deserialization as fast as possible*. For that reason, we use a simple, unoptimized serialization in our current CoDeSe implementation. Our serialization traverses the object graph reachable from a given reference much like the standard Java serialization. However, we implemented our own graph traversal rather than reusing the existing, highly optimized traversal from `java.io`, because having our traversal made it easier to develop and debug CoDeSe code generators. Our current CoDeSe serialization is usually a few times slower that the standard Java serialization, but we surprisingly noticed several examples when CoDeSe performs equally well or even much better than Java due to the optimized treatment of default and literal values in CoDeSe. Note that the direct goal of this optimized treatment is to obtain faster deserialization, and obtaining faster serialization is just a side benefit.

Our current CoDeSe serialization does not directly generate Java bytecode (or native code for the JNI code generator) but instead generates Java source code (or C code for the JNI code generator). Generating source code allows easier debugging and is readable by the user (similarly to serializing to XML rather than to a binary data format [23,40]), but it requires compilation. To estimate the benefit of directly generating Java bytecode, we wrote a code generator specialized for RBT that generates bytecode and found that it takes about the same time as generating source code but removes the compilation time.

We leave it as the future work to implement a general code generator that generates bytecode and to optimize CoDeSe serialization further. Note, however, that there are numerous cases where a system with a slower serialization but a faster deserialization is still better than system with a faster serialization but a slower deserialization. First, consider the scenario where an object graph is serialized (once) and deserialized (once or multiple times) in the same JVM run. The overall running time can be smaller even if serialization takes more time, as long as deserialization takes less time.

Second, *serialization is not on the critical path for some scenarios*, e.g., when an object graph is serialized in one JVM run but deserialized (much) later in different JVM run(s). As one example, consider the XRTQ checkpointing for debugging [41]. The most valuable asset in debugging is programmer's time, which is wasted when the programmer idles waiting for the program. XRTQ reduces the idle time by faster restoring the state of long-running Java programs. XRTQ uses serialization to capture (a part of) the Java heap at program points where the programmer expects to restore the state. While the state is captured in one JVM run, the programmer can interactively restore the state in many runs. The delay from when the state is captured to when it is first restored can be large, e.g., overnight. In such a scenario, a system can use the fastest possible serialization to on-line capture the state (potentially generating data rather than code as the serialization format), and then off-line transform the captured state to the format that allows the fastest possible deserialization. In fact, CoDeSe implementation already supports off-line transformation from stored data to stored code: one can use the standard Java `ObjectInputStream` to restore an object graph from a serialized file and then serialize this graph using `CodeseOutputStream`.

As another example of non-critical serialization, consider OCAT [16], which uses captured objects in test generation and execution. There is a big delay between generating a unit test (once) and re-executing it for regression testing (many times). During that delay, one can transform the XML files generated by OCAT into code whose execution can restore the captured objects much faster than XML parsing. In fact, our experiments in Section 4 use some XML files directly taken from the OCAT experiments and transform these files into the code generated by CoDeSe.

3.8 Limitations

While CoDeSe supports a large number of features offered by the standard Java serialization—including object streams, default serialization with the `Serializable` interface, and specialized serialization with private `writeObject` methods—CoDeSe currently does not support two features: specialized serialization with `replaceObject`/`putFields` and class evolution. The `ObjectOutputStream` class offers methods `replaceObject` (to allow trusted subclasses to replace some of the objects encountered during serialization) and `putFields` (to operate on the object that buffers fields to be written to the stream), but these methods are rarely used.

The standard library supports class evolution with a special field `serialVersionUID` that can be added to classes that implement `Serializable` to encode their version. The field is used to check the version of the class before loading it. If a class is evolved by adding/removing some fields, objects serialized with an old version of the class may still be deserialized with the new version of the class. CoDeSe cannot easily support full class evolution, although it supports one special case (if a class is evolved by only adding new fields at the end of the list of instance fields, and deserialization should set those new fields to their default values). To support full class evolution would require rewriting the code that CoDeSe generated, e.g., to change/remove/add some field assignments/offsets in the `setFields` methods as in Figure 3(b). In our intended uses of CoDeSe for testing, model checking, and debugging, evolution is not a big issue as (largely) the same code is run multiple times, with no (big) changes in class declarations. For other uses of serialization, where classes can have bigger changes, e.g., persisting objects in databases and using them much later, one would use serialization based on stored data and not stored code.

The current CoDeSe implementation increases memory consumption in a JVM, because all the classes used for deserialization remain in the JVM until it terminates. A JVM can unload a class only if its `ClassLoader` becomes unreachable [11]. To reduce memory consumption for CoDeSe would thus require writing a special class loader that would allow unloading the classes that are not needed in the future deserializations. We leave writing a class loader as future work.

4. EVALUATION

We performed a number of experiments to compare deserialization time when using the standard Java library and our CoDeSe implementation in several modes. The comparison also includes CoDeSe that generates JNI code (Section 4.5). First, we describe our experimental setup, including the object graphs on which we performed the experiments. Then, we discuss the timing results. All experiments were performed on a machine with a 4-core Intel Xeon 2.13GHz processor and 4GB of main memory, running Linux version 2.6.18, and Java HotSpot 64-Bit Server VM, version 1.6.0_10. (We have obtained similar results, although somewhat smaller speedup, using IBM J9 VM, version 1.6.0.) Our basic experiments disable bytecode verification because CoDeSe is intended to be used in scenarios where its code is trusted. Next, we discuss the result if the bytecode verification is enabled. Finally, we present the size of serialized object graphs.

	graph	#objects	#refs.	#prims.
Math	ArrayRVector	2	1	216
	CommathDir	118371	22134	244510
	CurveFitter	418	219	3478
	LOFunction	1	0	1
	LMOptimizer	418	219	2448
Collection	BinHeap	109	115	4
	BinHeapDir	1229045	1340273	47932
	DOrdMap	3774	6136	1886
	DOrdMapDir	2841720	5277480	1622238
	FastTreeMap	29	5	1
AspectJ	BcelCode	622	623	1242
	BcelCodeDir	111652	165065	218639
	BranchHandle	368	635	538
	BcelMethod	628	637	937
	InstFactory	302	1230	459
X10K	FRASDist	934	1566	1
	KMeans	217	348	1
	NQueensPar	371	632	1
	StructSpheres	278	446	1
	Runtime	60769	101096	242

Figure 4: Statistics for non-RBT object graphs.

4.1 Experimental Setup

We use three kinds of object graphs in our evaluation, one kind based on our running example and two kinds with states captured during the execution of real applications. First, we use RBT object graphs similar to the example from Section 2 but with trees of different sizes, denoted RBT N for N nodes. We create each tree by repeatedly calling a `put` method to insert N random values into the tree. Second, we use 15 object graphs from the OCAT experiments [16], provided to us by the OCAT authors. OCAT is a testing tool that captures graphs from actual executions, and we use five graphs each from executions of Apache Commons Collection, Apache Commons Math, and AspectJ. Third, we use five graphs from the same scenario as in the XRTQ study [41] that showed how deserializing the state of a compiler can be faster than re-executing the compiler from the beginning (with loading sources, parsing, etc.). Since the XRTQ infrastructure was not available to us, we used the X10K compiler, developed in our group, which translates X10 programs [36] to the K notation [35]. We ran X10K on several sample programs from the X10 distribution and on the X10 runtime, and captured the abstract syntax trees from X10K as object graphs.

Each object graph was serialized into two files, one using the Java format and one using the CoDeSe format. Figure 4 shows for each graph the number of objects, reference field assignments, and primitive field assignments generated by CoDeSe. (Note that some graphs seemingly have fewer references than objects, which would imply that they are disconnected, but in fact they use the private `writeObject` method for connecting these objects.) To be able to serialize and deserialize certain objects with the standard Java library, we first had to change the classfiles for those ob-

graph	#deserializations=1							#deserializations=2			#deserializations=10		
	SJL [ms]	CoDeSe [ms]	ratio	#t=2 [ms]	ratio t2/t1	#t=3 [ms]	ratio t3/t1	SJL [ms]	CoDeSe [ms]	ratio	SJL [ms]	CoDeSe [ms]	ratio
RBT 10	20	7	2.86	10	0.70	10	0.70	21	7	3.00	29	7	4.14
RBT 100	25	8	3.13	10	0.80	10	0.80	31	8	3.88	74	9	8.22
RBT 1000	67	10	6.70	10	1.00	12	0.83	126	11	11.45	209	21	9.95
RBT 10000	228	31	7.35	28	1.11	23	1.35	301	41	7.34	452	116	3.90
RBT 100000	572	223	2.57	135	1.65	94	2.37	1094	329	3.33	2246	1020	2.20
RBT 1000000	5220	3252	1.61	2157	1.51	1949	1.67	8979	5227	1.72	33385	17676	1.89
RBT 2000000	13819	9489	1.46	5881	1.61	5537	1.71	23043	13758	1.67	71116	43947	1.62
RBT 4000000	35068	19768	1.77	14628	1.35	13515	1.46	57832	30922	1.87	158546	92159	1.72
ArrayRVector	18	2	9.00	4	0.50	4	0.50	18	2	9.00	20	3	6.67
CommathDir	663	230	2.88	181	1.27	175	1.31	840	301	2.79	1577	705	2.24
CurveFitter	68	7	9.71	9	0.78	8	0.88	77	8	9.63	147	13	11.31
LOFunction	47	34	1.38	36	0.94	37	0.92	47	34	1.38	50	35	1.43
LMOptimizer	70	6	11.67	8	0.75	7	0.86	79	6	13.17	148	10	14.80
BinHeap	48	2	24.00	5	0.40	5	0.40	51	2	25.50	73	3	24.33
BinHeapDir	1474	1065	1.38	856	1.24	708	1.50	2253	1751	1.29	8239	5043	1.63
DOrdMap	130	7	18.57	7	1.00	8	0.88	186	9	20.67	353	24	14.71
DOrdMapDir	4987	4487	1.11	3439	1.63	2685	2.21	8157	7111	1.15	35185	24489	1.44
FastTreeMap	51	4	12.75	6	0.67	7	0.57	52	4	13.00	62	5	12.40
BcelCode	93	10	9.30	7	1.43	8	1.25	108	10	10.80	210	14	15.00
BcelCodeDir	556	189	2.94	116	1.63	70	2.70	661	274	2.41	1348	930	1.45
BranchHandle	199	26	7.65	29	0.90	29	0.90	222	29	7.66	390	51	7.65
BcelMethod	122	10	12.20	20	0.50	11	0.91	139	10	13.90	250	14	17.86
InstFactory	137	10	13.70	22	0.45	14	0.71	149	11	13.55	243	21	11.57
FRASDist	301	24	12.54	29	0.83	25	0.96	340	25	13.60	472	33	14.30
KMeans	250	19	13.16	23	0.83	21	0.90	262	19	13.79	374	21	17.81
NQueensPar	264	21	12.57	23	0.91	23	0.91	283	21	13.48	406	25	16.24
StructSpheres	244	19	12.84	22	0.86	22	0.86	260	20	13.00	380	23	16.52
Runtime	607	108	5.62	97	1.11	97	1.11	708	146	4.85	1298	443	2.93
geometric mean			6.48							6.75			6.70

Figure 5: Comparison of SJL and CoDeSe for a number of deserializations, and comparison of CoDeSe for a number of threads. Speedup is ratio of times in milliseconds.

jects to implement `java.io.Serializable`. CoDeSe, like some Java-to-XML serialization libraries [23,40], does not require that classes implement `Serializable`.

We wrote a small driver program that invokes deserialization from a given stream one or more times, and measures the execution time. Recall from the introduction that there are two general patterns for deserialization: (1) the same state is deserialized multiple times in one JVM run (typically for testing, e.g., in OCAT), or (2) one state is deserialized only once in a JVM run (typically for debugging, e.g., in XRTQ). The first deserialization for both Java and CoDeSe loads the stored information (data in case of Java, and code in case of CoDeSe) from a file, i.e., the underlying stream is a (buffered) `FileInputStream`. For CoDeSe, the driver need not do anything special for multiple deserializations; it only calls the same `readObject` to execute multiple times. For Java, however, we can improve the performance when we expect multiple deserializations. Indeed, our driver does not load the file multiple times but instead loads the file content only once and puts it into a `ByteArrayInputStream` such that all deserializations call `readObject` on this underlying stream (calling the `reset` method between deserializations). Finally, the driver measures the execution time of the appropriate `readObject` calls.

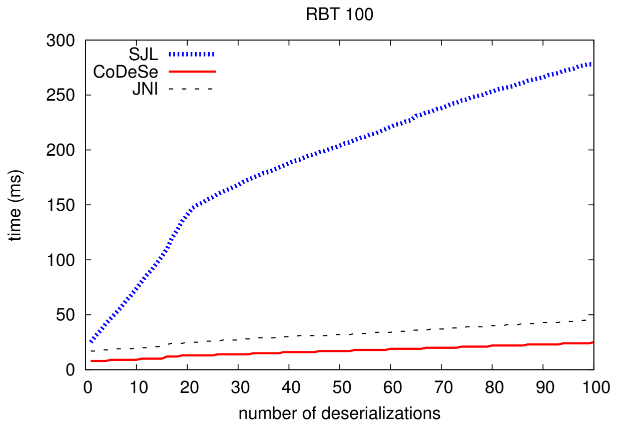
We point out that although each of our experiments explicitly deserializes only one object graph per execution, some of these graphs consist of many classes and their experiments effectively deserialize a number of different, smaller graphs in one execution. In particular, `CommathDir`, `BinHeapDir`, and `BcelCodeDir` include a number of graphs that represent different objects, many of them of different types.

To increase confidence in the correctness of our implementation, we checked it as follows. Each graph was serialized using (1) SJL and (2) CoDeSe. Then, we deserialized the graph (2) using CoDeSe and serialized it again using (3) SJL. Finally, we compared files (1) and (3).

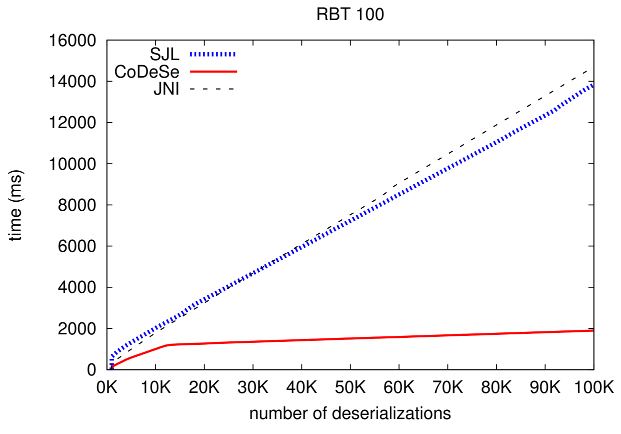
4.2 SJL vs. CoDeSe

Figure 5 shows a comparison of deserialization times using the Java library and the CoDeSe library. For each graph, we tabulate the time to deserialize it once (columns 2-3), twice (9-10), and ten times (12-13). The time is given in milliseconds. The columns 4, 11, and 14 show the speedup that CoDeSe achieves over SJL, computed as the ratio of deserialization times. *CoDeSe sped up deserialization in all cases*, although the graph characteristics differ (sizes ranging from small to large, and the type of fields as shown in Figure 4), and the number of deserializations ranges from one to ten. Section 4.4 discusses even larger number of deserializations for some graphs.

It is also interesting to point out that, for both SJL and CoDeSe, the deserialization time often scales super-linearly with respect to both the graph size and the number of deserializations. For example, consider the cases of RBT 1000 and RBT 10000 for #deserializations=1. Although the graph is ten times larger, the deserialization time went up only about three times (for both SJL and CoDeSe). As another example, consider RBT 1000 for #deserializations=1 and #deserializations=10. Although the number of deserializations went up ten times, the deserializations time went up only



(a)



(b)

Figure 6: Deserialization time (cumulative value) for RBT with 100 nodes with respect to the number of deserializations when the number ranges (a) from 1 to 100, and (b) from 1 to 100000.

about two times (for both SJL and CoDeSe). These results are the consequence of startup costs (loading data or class files, initializing classes, etc.) but also of JIT compilation as discussed in Section 4.4.

4.3 Parallel Deserialization

Recall from Section 3.3 that the CoDeSe serialization format based on code allows a simple parallelization of deserialization. Figure 5 (columns 5-8) shows how the deserialization time changes with the increasing number of threads used for deserialization. While there is some speedup, especially for larger graphs, the speedup is not nearly linear, and in fact going even from two to three threads sometimes results in a slowdown. (The machine on which we performed experiments has four cores.)

Our inspection shows that one reason for slowdown (or sub-linear speedup) is the overhead of creating threads and synchronizing on barriers, as expected. However, an even bigger reason is the CoDeSe phase for setting field values. While the field-setting phase is data-parallel, the threads read various parts of the same array that stores the objects (`objs`) and write to fields of various objects. This affects cache behavior because setting a large number of fields can effectively sweep the cache, and setting values from different threads can result in cache collisions. We leave it as future work to further improve scalability of parallel deserialization, and to consider parallel serialization.

4.4 Large Number of Deserializations

In a typical usage scenario of CoDeSe, we expect that an object graph be deserialized once or a relatively small number of times (say, up to ten or at most a few dozen times) in one JVM run. For that reason, Figure 5 shows up to ten deserializations. Nevertheless, we considered what happens if an object graph is deserialized a much larger number of times, and we found some interesting results that apply even for a small number of deserializations.

Figure 6 shows how deserialization time increases with the number of deserializations. We show plots for RBT with 100 nodes; the results are similar for other graphs. As expected, the time grows mostly linearly. However, both plots have a visible "knee" points where the linear rate of growth switches to a smaller value: Figure 6(a) has such a point for SJL and about 20 deserializations, and Figure 6(b) has such a point for CoDeSe and about 12000 deserializations. These are the points where just-in-time (JIT) compilation has happened, and the JVM optimized the code to run faster.

Typically a JVM optimizes a code unit, such as a method, after the unit is executed some number of times. During SJL deserialization, certain methods are repeatedly executed *for each object* being created. Hence, these methods can reach the threshold for JIT compilation with a relatively small number of deserializations. In contrast, during CoDeSe deserialization, each method is executed only once. Hence, these methods require a large number of deserializations to reach the threshold for JIT compilation.

Note, however, that *CoDeSe deserialization remains faster than SJL deserialization for all numbers of deserializations*, although the SJL deserialization code is JIT compiled much earlier than the CoDeSe deserialization code. In fact, the relative speedup that CoDeSe provides over SJL typically grows with the number of deserializations. It means that JIT compilers can produce more efficient code from the CoDeSe deserialization code than from the SJL deserialization code. This is understandable because the CoDeSe deserialization code is effectively a specialized version of the general SJL deserialization code. The specialization is performed by the CoDeSe code generator with respect to the given object graph; such an extensive specialization is unlikely to be performed by a JIT compiler.

The implication for the realistic usage scenario with a small number of deserializations is as follows. CoDeSe would be even faster than SJL if JVMs supported saving and loading JIT-compiled code between runs. While most widely used JVMs do not have any such support, IBM's J9 does have some support for loading ahead-of-time compiled code.

4.5 JNI

Recall from Section 4.5 that we developed a CoDeSe code generator to output C code that uses the JNI interface to create objects and set their fields. Figure 7(a) shows the deserialization time for JNI. For each graph, we tabulate the time to deserialize it once, twice, and ten times. In all cases, JNI was somewhat faster than SJL but much slower than

graph	JNI [ms]			Bytecode Verification				Disk Space		
				CoDeSe #d=1 [ms]		CoDeSe #d=10 [ms]		SJL	CoDeSe	JNI
	#d=1	#d=2	#d=10	BV	speedup	BV	speedup	size [KB]	size [KB]	size [KB]
RBT 10	25	25	26	17	2.43	18	2.57	0.6	48.0	14.0
RBT 100	17	17	20	17	2.13	18	2.00	3.0	60.0	51.0
RBT 1000	38	41	59	32	3.20	45	2.14	27.0	188.0	418.0
RBT 10000	50	74	225	159	5.13	242	2.09	265.0	1536.0	4096.0
RBT 100000	310	506	2072	1685	7.56	3147	3.09	2662.0	17408.0	40960.0
RBT 1000000	-	-	-	15758	4.85	31414	1.78	26624.0	168960.0	-
RBT 2000000	-	-	-	29939	3.16	62454	1.42	53248.0	337920.0	-
RBT 4000000	-	-	-	70050	3.54	145551	1.58	105472.0	675840.0	-

(a) (b) (c)

Figure 7: (a) Deserialization time using JNI; (b) Deserialization time with bytecode verification; (c) Disk space for serialized object graphs.

the default CoDeSe that generates Java code (see Figure 5 for comparison). We find that JNI is slower than CoDeSe because our C code properly uses the JNI function calls to communicate with the JVM, and these calls are reported to have a large overhead [18]. These JNI function calls preclude many compiler optimizations on the generated C code. Moreover, JNI code is compiled statically and not further JIT optimized by the JVM. In fact, looking back at Figure 6, we see that while JNI is faster than SJL for a smaller number of deserializations, JNI becomes slower for a larger number because the SJL code gets JIT compiled.

We performed a small limit study to see how fast the C code could get if it did not use the JNI abstraction but rather directly accessed the internals of JVM. For example, instead of using the JNI function `SetObjectField` to set some field value, our manually optimized C code directly manipulates the C pointers that point to the JVM representation of Java objects. Effectively, we inlined `SetObjectField` calls, breaking the abstraction but allowing many compiler optimizations on the C code. Our highly optimized code indeed became much faster than JNI but still not as fast as the JIT compiled Java bytecode from CoDeSe. This result attests to the very high quality of JIT compilers in modern JVMs.

4.6 Bytecode Verification

While the C code we generated in our limit study broke the JNI abstraction, the Java code that CoDeSe generates is always valid Java code. CoDeSe does not require any changes to JVM and works on top of any JVM that provides `sun.misc.Unsafe` or a similar class. However, when a JVM loads the code generated by CoDeSe, the JVM would still perform the regular bytecode verification that JVM performs to check validity of loaded code [11]. Because we assume that CoDeSe is used in a testing or debugging scenario where the CoDeSe code is trusted, we disabled bytecode verification by loading CoDeSe generated code through the `bootclasspath` rather than the usual `classpath`. Figure 7(b) shows the deserialization times that would be obtained with bytecode verification enabled, and the speedup provided by disabling bytecode verification. (The speedup is computed as the ratio of appropriate times from figures 7 (b) and 5.) While CoDeSe with bytecode verification is slower than without verification, it is still sometimes faster than SJL.

4.7 Space for Serialized Representation

The key metric for fast deserialization is the time needed to recreate object graphs from the serialized representation, but we also discuss the space needed to store the serialized representation. Figures 7 (c) and 8 show how much disk space is required to store (1) data for SJL serialization, (2) bytecode for the default CoDeSe code generator that produces Java, and (3) (only for RBT) native code for the CoDeSe code generator that produces C. We can see that CoDeSe requires several times more space than Java, and JNI requires even more than the default CoDeSe. While the increased space requirement of CoDeSe over SJL could be an important issue for storing many states or for sending the states over the network, it is not a significant issue for our intended scenarios of testing and debugging. Note that, although CoDeSe requires more disk space and thus has a higher loading time for the first deserialization than SJL, CoDeSe still has smaller overall time than SJL even for the first deserialization (and even more so for multiple deserializations in one JVM run).

	graph	SJL size [KB]	CoDeSe size [KB]	ratio
Math	ArrayRVector	1.8	60.0	0.03
	CommathDir	3482.0	9523.0	0.37
	CurveFitter	33.0	160.0	0.21
	LOFunction	1.9	60.0	0.03
	LMOptimizer	25.0	132.0	0.19
Collection	BinHeap	1.5	52.0	0.03
	BinHeapDir	12288.0	63488.0	0.19
	DOrdMap	45.0	264.0	0.17
	DOrdMapDir	37888.0	215040.0	0.18
	FastTreeMap	0.7	52.0	0.01
AspectJ	BcelCode	11.0	112.0	0.10
	BcelCodeDir	1536.0	11264.0	0.14
	BranchHandle	14.0	100.0	0.14
	BcelMethod	12.0	104.0	0.12
	InstFactory	10.0	76.0	0.13
X10K	FRASDist	16.0	96.0	0.17
	KMeans	5.7	52.0	0.11
	NQueensPar	8.2	60.0	0.14
	StructSpheres	6.7	56.0	0.12
	Runtime	692.0	4301.0	0.16
	geometric mean			0.11

Figure 8: Disk space for serialized object graphs.

5. RELATED WORK

Work related to CoDeSe can be split in three groups: optimizing serialization/deserialization, defining format to represent serialized object graphs, and using serialization/deserialization in testing techniques and tools.

Kamin et al. [3, 24] were the first to deploy run-time code generation to optimize *serialization*. Their approach generates a specialized version of the serialization method for a class `C` when the serialization is first performed on an object of type `C`, and later uses this specialized version when another object of type `C` is serialized. Mesquita [29] follows the same approach for Java, while Tansey and Tilevich [38] propose compile-time specialization for C++ version of MPI. The specialized code in all these approaches still produces *data* as the traditional serialization, whereas CoDeSe produces *code*. Effectively, all these approaches specialize the *serialization code with respect to the types* of the objects being serialized such that serialization generates stored data faster. In contrast, CoDeSe specializes the *deserialization code with respect to the values* in the objects being serialized such that deserialization restores the state faster.

Abu-Ghazaleh et al. introduced concepts of differential serialization [2] and deserialization [1] to optimize SOAP performance. The basic idea behind these techniques is to avoid full serialization/deserialization of each message and instead to serialize/deserialize just the parts of the message that differ from the previous message that has been sent/received. These techniques showed significant speedup when consecutive messages are similar but unfortunately had a large overhead when messages are dissimilar. CoDeSe does not consider any similarity among consecutive states and speeds up deserialization over the standard Java library both when states are similar and dissimilar.

Increase in the amount of data being transfered over the networks has led to the development of several protocols that target performance improvement of the data transfer. Two widely used protocols are Google Protocol Buffers [34] and Apache Avro [5]. Both protocols are based on the same principles—statically generate code that serializes/deserializes data to/from different type of streams and remove the meta information from the serialized representation. The user is responsible to specify the format of serialized object graph in a domain specific language (DSL). These protocols are able to generate code in different programming languages from the same DSL. Removing the meta information reduces the size of the serialized object graphs. In addition, Protocol Buffers does not include null values, but CoDeSe also does not include default value even for primitive fields. In contrast to CoDeSe, these protocols require the user to provide a DSL description of the format used for serialized graphs, which can be a non-trivial task. While these protocols focus on the message size for transfer over the network, our primary focus is fast deserialization for applications that deserialize the state once or multiple times.

XStream [40] is a Java library similar to the standard `java.io` serialization/deserialization, but XStream stores the state of the program in the XML format rather than binary. Also, it offers a more flexible API and does not require the classes to be serialized to implement the `Serializable` interface. Similar to XStream, JSX [23] is another library for Java that serializes the object states to the XML format. In contrast to these libraries, CoDeSe specifies a new format for serialized object graphs based on executable code. Also, the default CoDeSe code generator that outputs Java source code is as readable as XML file.

JSON [22] is a text-based approach, subset of JavaScript syntax, for data interchange that can be seen as an alternative to XML. As opposed to XML that has to be parsed, JSON file can be directly loaded (without parsing, using `eval`) into JavaScript code. The key issue is that JSON standard does not support object references. An available extension [7] that does support references performs parsing of the file, similar to XStream approach for XML. CoDeSe supports references and does not require parsing.

As mentioned in the introduction, many testing techniques use serialization and deserialization. Throughout the paper we already described parts of several techniques such as OCAT [16], XRQT [41], and JPF [39]. We next describe several other techniques.

Elbaum et al. [8, 9, 19], Kumar and Baar [25], and Orso et al. [20, 31] propose techniques that use serialization and deserialization for building unit tests from system tests. Their techniques capture interaction between modules during the execution of system tests to later replay for unit tests. Their capture/replay phases use serialization/deserialization for selected parts of the program state. Some techniques use XStream [40] extended to serialize/deserialize static fields. CoDeSe, as SJL and XStream, currently does not serialize/deserialize static fields. However, adding this extension to CoDeSe would be easy.

Artzi et al. [4] and Leitner et al. [26, 27] propose techniques that generate tests that reproduce system failures after the system crashes. ReCrash [4] proceeds in two phases: monitoring and test generation. In the monitoring phase, the techniques keep information about methods being called and arguments that were passed to these methods. If the system crashes, it uses these "stacks" of method calls to generate test methods for each test in the "stack". To reinvoke the method, it restores the state of the arguments and the receiver. Rather than capturing the pre-state at the method entry, similar results can be obtained by capturing the state at the point of the crash but then restoring this state as the pre-state at the method entry [27]. Luo et al. [28] extend ReCrash for multithreaded code. All these techniques involve deserializing Java states and could benefit from faster deserialization provided by CoDeSe.

Barnat et al. [6] describe efficient model checking of the applications that require large amount of states which cannot be stored in the main memory. In these cases, the states are written to external memories and restored when they are needed. CoDeSe could be used to speed up deserialization of the states.

Hruba et al. [13] explore the use of bounded-model checking to enable self healing. The exploration is performed after the suspicious state is detected. The recorded state has to be reconstructed in the model checking tool to enable the exploration. Although they use re-execution to reconstruct the state, storing/restoring through serialization/deserialization is an alternative, and CoDeSe could speed up deserialization.

6. CONCLUSION

We proposed CoDeSe as a novel format for representing serialized object graphs that can substantially improve the performance of deserialization. Unlike the existing approaches, which use a format based on data to represent serialized object graph, CoDeSe uses a format based on code. During serialization, CoDeSe generates code whose execution restores the state, and deserialization is simply the execution of this code. We implemented CoDeSe in Java and performed a number of experiments, including with states generated by real applications. CoDeSe provides on average more than 6X speedup over the highly optimized deserialization from the standard Java library. Our new format also allows a simple parallel deserialization that provides additional speedup over the sequential CoDeSe for larger states. In the future we plan to explore more efficient parallel deserialization and serialization.

Acknowledgements

We thank Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang for providing the data from their OCAT study, Matt Kirn for sharing his code that automatically adds the `Serializable` interface to classes (necessary for serialization in SJL), and Dmytro Suvorov and Vilas Jagannath for initial discussions about this work. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-0746856 and CNS-0958199, and by IBM under an X10 Innovation Grant.

7. REFERENCES

[1] N. Abu-Ghazaleh and M. J. Lewis. Differential deserialization for optimized SOAP performance. In *SC*, 2005.

[2] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential serialization for optimized SOAP performance. In *HPDC*, 2004.

[3] B. Aktemur, J. Jones, S. N. Kamin, and L. Clausen. Optimizing marshalling by run-time program generation. In *GPCE*, 2005.

[4] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *ECOOP*, 2008.

[5] Apache Avro home page. `http://avro.apache.org/`.

[6] J. Barnat, L. Brim, and P. Simecek. Cluster-based I/O-efficient LTL model checking. 2009.

[7] Dojo home page. `http://dojotoolkit.org/`.

[8] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *FSE*, 2006.

[9] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *TSE*, 2009.

[10] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, 1997.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2005.

[12] T. Gvero, M. Gligoric, S. Lauterburg, M. d'Amorim, D. Marinov, and S. Khurshid. State extensions for Java PathFinder. In *ICSE Demo*, 2008.

[13] V. Hrubá, B. Krena, and T. Vojnar. Self-healing assurance based on bounded model checking. In *EUROCAST*, 2009.

[14] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, 2001.

[15] Java PathFinder (JPF) home page. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[16] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object capture-based automated testing. In *ISSTA*, 2010.

[17] Java class file format. `http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html`.

[18] Java Native Interface programmer's guide and specification. `http://java.sun.com/docs/books/jni/html/fldmeth.html`.

[19] M. Jorde, S. G. Elbaum, and M. B. Dwyer. Increasing test granularity by aggregating unit tests. In *ASE*, 2008.

[20] S. Joshi and A. Orso. SCARPE: A technique and tool for selective capture and replay of program executions. In *ICSM*, 2007.

[21] Java object serialization specification. `http://download.oracle.com/javase/6/docs/platform/serialization/spec/serialTOC.html`.

[22] JSON home page. `http://www.json.org/`.

[23] JSX home page. `http://jsx.org/`.

[24] S. Kamin, L. Clausen, and A. Jarvis. Jumbo: Run-time code generation for Java and its applications. In *CGO*, 2003.

[25] P. Kumar and T. Baar. Using AOP for discovering and defining executable test cases. In *Ershov Memorial Conference*, 2009.

[26] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test cases. In *ESEC/FSE*, 2007.

[27] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol. On the effectiveness of test extraction without overhead. In *ICST*, 2009.

[28] Q. Luo, S. Zhang, J. Zhao, and M. Hu. A lightweight and portable approach to making concurrent failures reproducible. In *FASE*, 2010.

[29] L. B. Mesquita. Faster Java serialization. `http://jserial.sourceforge.net/index.html`.

[30] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, 2008.

[31] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with JINSI. In *WODA*, 2006.

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[33] R. Pelanek. Typical structural properties of state spaces. In *SPIN Workshop*, 2004.

[34] Protocol Buffers home page. `http://code.google.com/apis/protocolbuffers/`.

[35] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 2010.

[36] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. Report on the programming language X10, 2010.

[37] B. Shah, P. R. Rao, B. Moon, and M. Rajagopalan. A data parallel algorithm for XML DOM parsing. In *International XML Database Symposium*, 2009.

[38] W. Tansey and E. Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *IPDPS*, 2008.

[39] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *J-ASE*, 2003.

[40] XStream home page. `http://xstream.codehaus.org/index.html`.

[41] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of Java software using context-sensitive capture and replay. In *ESEC/FSE*, 2007.