Optimizing Marshalling by Run-Time Program Generation*

Barış Aktemur[1], Joel Jones[2], Samuel Kamin[1], and Lars Clausen[3]

[1] University of Illinois at Urbana-Champaign, USA
{aktemur, kamin}@cs.uiuc.edu
[2] University of Alabama, USA
jones@cs.ua.edu
[3] State's Library, Aarhus, Denmark
lc@statsbiblioteket.dk

Abstract. Saving the internal data of an application in an external form is called *marshalling*. A generic marshaller is difficult to optimize because the format of the data that will be marshalled is unknown at the time the marshaller is implemented. On the other hand, efficient marshallers can be written for specific kinds of data. In this paper we use run-time program generation (RTPG) to produce specialized marshallers. We use Jumbo, a Java compiler supporting programmer-specified RTPG. We show that RTPG is easily employable. Speedups in order of magnitude can be achieved in some cases. We study the case where the data consist of a large number of objects of a single class and the case where there are objects of many classes. In the latter case, "just-in-time" heuristics allow us to limit RTPG costs and gain considerable speedups.

1 Introduction

Marshalling is the term used for saving the internal data of an application in an external form. Once marshalled, objects can be passed to other applications. Java RMI (remote method invocation) and CORBA are examples of systems which marshal data for transmission to remote machines. Another term for marshalling is *serialization*. The reverse process is called *unmarshalling*.

Serialization generally involves writing large amounts of data, and so is often a performance bottleneck. (According to [1], Java serialization accounts for 25–65% of a remote method invocation.) For any particular type of data, it can be heavily optimized. However, optimizing a general-purpose marshaller is difficult because the format of the data to be marshalled is not known at compile-time. Such marshallers are guided by a description of the data that becomes available only at run-time; it is provided either by the client of the marshalling code, or, as in the case we consider here, by the language's reflection mechanism.

Run-time program generation (RTPG) is the use of programs that write other programs at run-time. RTPG can produce efficiencies by taking advantage

* Partial support for this work was received from NSF under grant CCR-0306221.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 221–236, 2005.

of information not known at compile time. Since this is precisely the situation we have just described, marshalling is a natural application for RTPG.

Our research group has developed Jumbo [2,3], a compiler for Java that incorporates an easy-to-use run-time program generation mechanism. Jumbo is distinguished by its implementation strategy [4] and by its consequent generality: virtually any Java program can be generated at run-time. This makes Jumbo particularly easy to learn and use. Thus, we believe it can make the writing of run-time program generators a routine matter [5,6].

In this paper, we demonstrate the practicality of RTPG by applying Jumbo to the problem of optimizing marshalling in Java [7,8,9]. Interestingly, the implementers of the generic marshaller in Sun's standard Java library (`java.io.ObjectOutputStream`) thought its efficiency so important that critical parts of their implementation are written in C++. This takes this class beyond the routine and has the specific drawback that the code cannot be verified. It also renders the implementation unsuitable as a starting point for our experiment. Hence, we start from a pure Java implementation of serialization, provided by Kaffe [10]. We demonstrate that run-time program generation is easy to employ — requiring no more skill than ordinary programming — and can deliver very substantial speedups relative to the pure Java code. (We were not specifically attempting to catch up with Sun's implementation, but we have done so in some cases; we discuss this in section 7.)

On standard marshalling benchmarks — mainly long arrays or lists — a straightforward use of RTPG speeds up the Kaffe implementation by an order of magnitude. Cases involving many classes are more difficult because the cost of the run-time program generation itself cannot be so readily amortized; an adaptive method similar to that used in just-in-time compilers can be employed.

Our contributions in this paper are:

- Demonstrating that, with Jumbo, obtaining generative code based on non-generative code is straightforward.
- Showing significant speedups for marshalling with RTPG.
- Showing that adaptive methods can be applied to reduce the cost of run-time compilation.

The paper is structured as follows. In section 2, we discuss marshalling in Java in more detail and give some ideas about where RTPG might help. Section 3 introduces Jumbo, and section 4 shows how Jumbo can be used to implement the suggestions made in section 2. Section 5 gives performance comparisons between Kaffe and Jumbo for the benchmark cases — large, homogeneous and near-homogeneous collections, and heterogeneous collections. In section 6, we discuss usage of "just in time" program generation to reduce the cost of run-time compilation for heterogeneous data. In section 7, we briefly return to Sun's implementation that uses native code and ask two questions: Can our safe, run-time-generated code compete with Sun's implementation, and can we use RTPG to produce further optimizations of that code? Finally, section 8 reviews related work and section 9 presents our conclusions.

2 Marshalling in `Java`

Java provides a simple API for serialization. A Java programmer doesn't need to write any serialization code, but must simply declare her classes to implement the empty interface `java.io.Serializable`. If a class implements this interface, an instance can be marshalled by passing it to `java.io.ObjectOutputStream`'s (OOS) `writeObject()` method.

Sun provides a specification of serialization [11], and an implementation. However, that implementation uses native methods, written in C/C++, to gain efficiency. Therefore, it is not appropriate for our experiment. An implementation in pure Java[1] is provided by Kaffe [10]; we start our study there.

Throughout this paper we refer to Sun's and Kaffe's implementation as Sun OOS and Kaffe OOS, respectively. The implementation for marshalling which uses RTPG is referred as Jumbo OOS. (Actually there are two versions of Jumbo OOS, but it will be clear from the context to which one we're referring.) When it doesn't matter which OOS we're referring to, we just say OOS.

We now explain Java serialization in detail, to highlight the places that can be optimized by RTPG. The serialization format is roughly as follows: For each object, first write a descriptor for its class and then write the object's fields; primitive fields are written directly, and object fields are written recursively using the same format. To prevent outputting multiple copies of class descriptors or objects – and to avoid infinite loops – each class and object is assigned an id number, or *handle*; every class and object written is stored in a hashtable the first time it is seen, and only its handle is output on subsequent sightings. The pseudo code below outlines Kaffe OOS's `writeObject()` method.

```
writeObject(obj) { //method in Kaffe OOS
  if obj is null {
    writeNull
  } else if obj was already written { // look up the object in the hashtable
    write object handle
  } else if obj is an instance of Class or String {
    write obj according to the specification for that particular case
  } else if the object is an Array {
    for each element i in obj
      writeObject(i) //a recursive call
  } else{ // first write class description of object
    if class of obj was already written
      write class handle
    else
      writeObject(class of obj) //recursive call to serialize class descriptor
    // then write content of object
    if obj is Serializable {
      for each classDescriptor in the class hierarchy of obj
        for each field in the classDescriptor
```

[1] Actually there is one call to a native method, to test whether a class has a static initializer. This test is not available in the reflection API [10].

```
        if the field is primitive
          writePrimitive(field)
        else
          writeObject(field) // recursive call
    } else { throw Exception("obj is not serializable") }
  }
}
```

To summarize, each object is passed through a set of checks: Is the object null? Was it already written to the stream? Is it an array? Was its class descriptor already written? Is it `Serializable`? Finally, for each class descriptor in the inheritance hierarchy of the object, we find the fields of that class. For each field, if it is primitive, we write the actual value directly to the stream. Otherwise, we marshal it by making a recursive call. Note the use of reflection in the above, using class descriptors to discover the fields of the class.

We can optimize the serialization of objects of any class by generating a marshaller specific to it when we first see an instance of that class. After the specialized marshaller is generated, it can be used to serialize subsequent instances. With this alteration, the general marshalling procedure becomes:

```
writeObject(obj) { //method in Jumbo OOS
  if obj is null
    writeNull
  else if obj was already written // look up the object in the hashtable
    write object handle
  else{
    // look for specialized marshaller in the hashtable
    marshaller = getMarshallerFor(class of obj)
    if marshaller is not null // marshaller is found
      marshaller.write(obj)
    else if obj is an instance of Class or String
      // ... as above
      if obj is Serializable {
        // generate specialized marshaller and put it into hashtable
        marshaller = ProgGen.generateMarshallerFor(obj)
        storeMarshaller(marshaller)
        // ... as above
      }
  }
}
```

The bold faced lines in the code above show when to look for a specialized marshaller and when to generate it. As a technical point, the reader will note that a specialized marshaller is not used for marshalling right after it is generated. This is because the generated code writes only the handle of the class, but the class descriptor needs to be written the first time an object of the class is seen.

We now introduce Jumbo. Readers familiar with it can skip the next section. In section 4, we continue the present discussion by showing how to go from the Kaffe code for serialization to the Jumbo code.

3 Jumbo

Jumbo [3,5] is a staged compilation system for Java, allowing run-time program generation. It provides a high degree of programmer control, source level specification, and binary-level operation.

In Jumbo, the programmer specifies code to be generated at run-time by placing it within special quotation brackets: `$<` and `>$`. From the programmer's point of view, these brackets behave very much like ordinary string quotes, but the values represented are of type `Code`, not string, and ordinary string operations cannot be applied. For this to work, the enclosed piece of program cannot be arbitrary, but must be a parsable fragment. The effect of this restriction is that these fragments can be partially compiled, with the result that no external compiler has to be invoked at run-time to generate code. Since many computers have a Java run-time, but no compiler, this is an important practical feature.

A quoted Java fragment can have *holes* that will get filled with `Code` values not known at code-writing time. The syntax for holes is backquote (`‘`) followed by a syntax category, followed by a Java expression of type `Code` in parentheses. Consider

```
public Code infiniteLoopGen(Code body){
  return $< while(true){
              ‘Stmt(body)
            } >$;
}
```

The call `infiniteLoopGen($< if(i == 3) break; i++; >$);` would give us `Code` equivalent to:

```
while(true){
  if(i == 3)
    break;
  i++;
}
```

This code can now be used in a context where `i` is defined.

For expressions of primitive type, there is a second kind of anti-quotation, one which evaluates the expression at program-generation time and then inserts the value into the generated code as a constant. For example, `‘Int(x)` means that x is an `int` variable and its *current* value is to be inserted into the enclosing `Code` (this is called *lifting* [12]).

`Code` is the main class in the Jumbo implementation. A `Code` value represents the *partially* compiled version of a program fragment and is represented as a method. Its argument is the information about the usage context of that fragment that is needed to fully compile the fragment; its result is the virtual machine code thus calculated. Because it is a method, this program fragment is represented as virtual machine code, rather than as source or as a syntax tree.

Detailed information on Jumbo is available in [3,5,2]. Jumbo can be obtained at `loome.cs.uiuc.edu/Jumbo/index.php`.

4 Jumbo Code for Marshalling

In section 2, we showed how to make use of program generation in Jumbo OOS. In this section we discuss how to write the specialized marshaller generator using Jumbo. We have implemented a class, called `ProgGen`, which produces the marshallers. Before we explain `ProgGen`, let's look at the specialized marshaller that would be produced for the following class, representing a linked-list node:

```
public class Node implements Serializable{
  int data;
  Node next;
}
```

Its generated marshaller would be:

```
public class NodeMarshaller implements Marshaller {
  JumboObjectOutputStream oos;
  Field[][] fields;
  int handle;

  public void init(JumboObjectOutputStream oos,
                   Class clazz, int handle) {
    this.oos = oos;
    this.handle = handle;
    ... // initialize fields[][] here - omitted
  }

  public void write(DataOutput stream, Object obj) {
    // Write the OBJECT tag and class handle to the stream
    // These magic numbers are defined in Sun's specification.
    stream.writeByte(115);
    stream.writeByte(113);
    stream.writeInt(handle);
    // write the 'data' field
    stream.writeInt(fields[0][0].getInt(obj));
    // send the 'next' field to Jumbo OOS to have it serialized
    oos.writeObject(fields[0][1].get(obj));
  }
}
```

Note that Jumbo generates byte code – *not* source code. We have given source code for readability: the byte code generated is just what would be produced by a Java compiler if presented with this source code.

When compared with the original OOS, the specialized marshaller is much simpler. The `next` field of `Node` will also be serialized via the specialized marshaller (provided that its run-time type is `Node`). The marshalling process will end when `next` is a null pointer or an already serialized object.

`ProgGen` is obtained by a fairly straightforward massaging of the Kaffe OOS. Basically, `ProgGen` and Kaffe OOS have code in one-to-one correspondence. However, `ProgGen` does not write data into a stream like Kaffe OOS does. Instead, it

forms `Code` which does that job. To illustrate, let's examine the `writeFields()` method of Kaffe OOS. This is the method that actually writes the fields of an object.

```
private void writeFields(Object obj, ObjectStreamClass osc){
  ObjectStreamField[] fields = osc.fields;
  String field_name;
  Class type;
  for (int i = 0; i < fields.length; i++){
    field_name = fields[i].getName();
    type = fields[i].getType();
    if (type == Boolean.TYPE)
      realOutput.writeBoolean(
        getBooleanField(obj, osc.forClass(), field_name));
    else if ... // check for other primitive types
    else        // non-primitive
      writeObject(getObjectField(obj, osc.forClass(),
                  field_name, fields[i].getTypeString ()));
  }
}
```

This method first gets all the fields in a class descriptor. Then, by using each field's descriptor, it fetches the value of the field from the object. This is done in `get`*X*`Field()` of OOS, which uses the `getField()` method below (exception-handling is omitted here for clarity):

```
private int getIntField (Object obj, Class klass, String fname) {
  Field f = getField(klass, fname);
  return f.getInt(obj);
}

Field getField (Class klass, String name) {
  final Field f = klass.getDeclaredField(name);
  AccessController.doPrivileged(new PrivilegedAction() {
    public Object run() {
      f.setAccessible(true);
      return null;
    }});
  return f;
}
```

This work is done for each object, even if another object of that class was already written. We shouldn't have to find the field specifiers and field types each time. Instead we can generate code with these values built in:

```
private Code writeFields(ObjectStreamClass desc, int hier) {
  ObjectStreamField[] fieldDecls = desc.fields;
  Code c = $< ; >$;
  for (int i = 0; i < fieldDecls.length; i++){
    Class type = fieldDecls[i].getType();
```

```
      if (type == Boolean.TYPE)
        c = $< `Stmt(c)
                stream.writeBoolean(
                  fields[`Int(hier)][`Int(i)].getBoolean(obj));
              >$;
      else if ... // other primitive types
      else // non-primitive type. write the field via Jumbo OOS
        c = $< `Stmt(c)
                oos.writeObject(
                  fields[`Int(hier)][`Int(i)].get(obj));
              >$;
    }
    return c;
  }
```

In the code above, `fields[][]` holds the field specifiers. The first index corresponds to the position of the class descriptor in the hierarchy, and the second index corresponds to the position of the field in that class descriptor. Note that the method requires `hier` as an argument. It doesn't need the `Object obj` parameter anymore, in contrast to the implementation of `writeFields` in Kaffe OOS. The code shows that if the field is non-primitive, it is passed to the Jumbo OOS to be written. In fact, we keep a one-element cache in the specialized marshaller associated with each non-primitive field; if the run-time type of the field is the same as the one in cache, we call the associated specialized marshaller without passing the object to Jumbo OOS. This saves us from the hashtable lookup that would occur in Jumbo OOS. If there is a cache miss, we pass the object to Jumbo OOS, it does a hashtable lookup, writes the object and then we update the cache. We do not give this code because of space limitations.

After we have the methods that return `Code` to serialize an object, we need to generate the `init` method[2], which will set up the data in the generated marshaller. In particular, this method is where the class handle is assigned to a data member of the serializer and where the `fields[][]` matrix is set. Note that this happens only once per generated serializer. This initializer method is constructed using code pieces from Kaffe OOS. Therefore writing this method is again straightforward, and to save space we don't provide the source code here.

The generated marshallers implement an interface called `Marshaller`, which defines the methods `init` and `write`. Interfaces, or abstract classes, are normally required in Java when ordinary code is to call generated code [3,5,2].

5 Performance

When using RTPG, the cost of run-time program generation must be taken into account. For this cost to pay off, we need to use the generated program a lot; that

[2] Java doesn't provide the ability to pass arguments to the constructors of dynamically loaded classes, so the class can only have a zero-argument constructor [3,13]. Thus we define a normal method, `init`, and call it right after the object is created via the zero-argument constructor.

is, we need to marshal a large data set. Still, the running time of the generated code — excluding compile time — is a useful quantity to know, because it gives the upper limit of speed-up (to which the actual speed-up will converge, over time). In this section, we give the performance of specialized marshallers, both including and excluding the cost of run-time compilation.

The performance of marshalling code is highly dependent upon the properties of the data being marshalled. Furthermore, it is not clear what should count as a "realistic" workload for marshalling. Large data sets — which are the ones we most care about, since these will be the most time-consuming to marshal — are likely to consist of large numbers of a few kinds of objects; this would be characteristic of video or audio streams, for example. On the other hand, most data in Java consists of objects of many different types. From the point of view of run-time program generation, these two scenarios have very different performance characteristics. Accordingly, we show benchmarks of both kinds. Specifically, we start by marshalling large, homogeneous collections of a class called `Dummy`, which has several fields. Then we test a linked-list class, and a class similar to `Dummy`, but with fields which can contain either of two types of objects (one a subclass of the other). After showing benchmarks for these homogeneous and near-homogeneous collections, we discuss a non-homogeneous data set, containing objects of 66 different classes.

Table 1. Performance table for `Dummy` class. Crossover point is 250 objects

Object Count	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	Kaffe Jumbo	Kaffe Jumbo+comp.
1000	30000	6.6	26.9	152.9	23.1	5.68
10000	300000	121.1	140.6	1545.0	12.75	10.99
20000	600000	257.8	277.0	3121.4	12.1	11.27

These benchmarks are run as follows: All the tests are executed on a Linux Debian, AMD Athlon XP 1700+ machine with 900MB memory. The timings are in milliseconds. We use HotSpot as the Java Virtual Machine, which is the most popular JVM[3]. When running a test, we first marshal a substantial number of objects to give the virtual machine time to *warm up*. During this time, the JVM loads classes and performs just-in-time optimization. Our experience has shown that this approach gives more consistent results. After warming up the JVM, we begin the test. We create a certain number of serializable objects, then pass the objects to the OOS's and measure the time spent. We call this *a benchmark*. After a benchmark is done, we discard the objects and OOS's —together with the hashtables they contain— and run another benchmark with a different number of objects. Thus, each benchmark begins with the Jumbo API and OOS's loaded

[3] Performance measurements with IBM's JVM [14] actually show significantly better speedups for Jumbo, but space prevents us from including these timings.

and optimized, the specialized marshallers not generated. In the tables below, each row represents a benchmark.

5.1 Homogeneous and Near-Homogeneous Data

Table 1 gives the results for marshalling objects of the `Dummy` class:

```
public class Dummy implements Serializable {
    Simple simple1;
    Simple simple2;
    int id;
}

public class Simple implements Serializable {
    int id;
}
```

The Jumbo OOS column does not include the run-time compilation cost, but Jumbo+compilation does. We have shown timings for marshalling 1000 to 20000 objects. The "Bytes written" column gives the size of the data written to the output stream. Jumbo OOS is at least 12 times faster than Kaffe OOS, when run-time generation cost is not included.[4]

Table 2. Performance table for linked-lists of `Dummy` objects. Each list has fifty nodes.

Number of lists	Bytes written	Jumbo OOS	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo	Kaffe / Jumbo+comp.
10	19363	6.7	48.9	145.0	21.42	2.96
50	84479	45.1	71.9	723.9	16.04	10.06
100	186877	107.7	131.3	1496.6	13.88	11.39
150	246075	115.8	135.4	2145.9	18.52	15.84
200	352161	144.6	174.4	2896.0	20.02	16.60

In our next test, we marshal linked-lists of Dummy nodes (same as `Node` class, but with data of type `Dummy`). Each linked list has 50 nodes. Jumbo OOS is up to 20 times faster than Kaffe OOS in this test. (See Table 2.)

Inheritance affects the cost of marshalling because it requires that we test the type of each field and not simply call the marshaller for the declared type of the field[5]. In the previous benchmarks, we did not marshal any objects whose classes had subclasses; thus, the actual type of every marshalled object was the

[4] The crossover points we give were determined by direct observation, not by interpolation from the presented data. We have omitted the timings for smaller data sets for lack of space.

[5] Remember that to eliminate some hashtable lookups, we associate a one-element cache with each field. See Section 4.

same as its declared type, and, in particular, the one-element cache always held the right class. For the next benchmark (Table 3), we marshal `Dummy` objects, but allow the fields of type `Simple` to contain either a `Simple` or a `SimpleChild` object, determined randomly. The `SimpleChild` class is shown below.

```
public class SimpleChild extends Simple{
    int otherId;
}
```

Table 3. Performance table for `Dummy` objects, allowing the fields to be either `Simple` or `SimpleChild`. Crossover point is 280 objects.

Object Count	Bytes written	Jumbo OOS	Jumbo + comp.	Kaffe OOS	Kaffe / Jumbo	Kaffe / Jumbo+comp.
1000	30136	9.9	55.3	154.7	15.55	2.80
10000	334320	136.3	167.2	1637.0	12.0	9.79
20000	641548	285.1	303.9	3237.3	11.35	10.65

5.2 Non-homogeneous Data

Data commonly consist of many objects of a variety of classes. This has a significant effect on the performance of our code because it implies a lot more classes being generated and therefore a lot more program generation time. In this section we examine the behaviour of Jumbo OOS on such data.

Table 4. Performance table for heterogeneous data. The objects come from a total of 66 classes.

Object Count	Bytes written	Jumbo OOS	Jumbo + comp.	Kaffe OOS	Kaffe / Jumbo	Kaffe / Jumbo+comp.
13210	128140	76.5	1504.1	1830.0	23.92	1.22
39630	372578	239.5	1690.9	5486.0	22.9	3.24
66050	617016	368.2	1837.9	9248.0	25.11	5.03
92470	861454	524.3	1899.5	12789.2	24.39	6.73
118890	1105892	657.4	2065.5	16499.7	25.09	7.99

For this purpose, we serialize `Code` objects. `Code` is the type of partially-compiled program fragments, as described earlier. In total, the `Code` objects indirectly touch 13210 objects, from 66 classes; 127 kilobytes were written to the stream. The timings are given in Table 4. We start by marshalling just one `Code` object, and increment by two on each row (i.e. marshal the object two more times than on the previous row). In this test, Jumbo OOS is faster than Kaffe OOS by approximately 25 times, when the cost of program generation is not counted. However, when code generation time is counted, the improvement relative to the Kaffe OOS goes down to about 1.22 in the worst case. The speed-up will approach 25 as the size of the data set increases, but it only achieves an eight-fold increase on the largest data set we tried.

The generated code shows much less speed-up than for the homogeneous case. Recall that the crossover point when marshalling `Dummy` objects was about 250 objects; now it is about 10500 objects. The problem, of course, is that we are generating code for many classes that have a small number of instances. We discuss this issue in the next section.

6 Just-in-time Program Generation

When marshalling heterogeneous data like `Code`, many classes are represented by only a few objects, and the cost of generating the marshalling code for those classes is not repaid. Our analysis of the test with heterogeneous data showed that only 14 out of the 66 classes allocated more than 250 objects. (Recall that, for `Dummy` objects, the crossover point was 250 objects.) Clearly, the remaining 52 classes will create a significant drag on the overall marshalling process.

To test the hypothesis that avoiding code generation for classes with few objects will yield better results, we ran a set of tests using varying *threshold* values: For each threshhold value, we generated code only for those classes which produce at least that many objects in the benchmark. This depends upon our having counted the number of objects for each class beforehand, so this does not represent a viable implementation strategy; we are only attempting to prove our hypothesis. We see (Table 5) that at a threshold value of 100, the generated code produces nearly a four times speedup over Kaffe OOS (compared to 1.22 fold speedup when all marshallers are generated). Note that, even at the optimal threshold value of 100, the speedup we can obtain in this situation is much less than we did with the simpler, homogeneous collections, because (1) the cost of run-time compilation is great due to the large number of classes and (2) many objects are marshalled by non-generated code.

Table 5. Performance comparison when threshold value is used. Marshallers are generated only for classes known to have more than threshold number of instances.

Threshold	Object Count	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo+compilation
20	13210	659.5	1861.2	2.82
60	13210	527.6	1871.3	3.54
100	13210	482.9	1872.0	3.87
140	13210	515.6	1869.9	3.62
180	13210	551.7	1855.4	3.36
300	13210	592.8	1871.0	3.15
400	13210	706.5	1868.1	2.64

In this experiment, the number of instances of each class was known prior to marshalling. What shall we do when we don't know that? The situation is similar to JIT compilation [15]. HotSpot keeps track of method calls and when a method is called a certain number of times, it is optimized.

Following this idea, our second version of the marshaller counts the number of objects marshalled. Once it has reached the threshold value, it generates specialized code and uses that for subsequent objects of the class. Note that this version will be slower than the previous one, because all objects marshalled prior to reaching the threshold value are marshalled by non-generated code. The results are shown in table 6. Here, we don't reach the previous speedup factor, but instead reach 3.16 (again with a threshold value of 100).

Table 6. Marshalling 13210 objects, with different threshold values

Threshold	Object Count	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo+compilation
20	13210	920.3	1851.6	2.01
60	13210	669.5	1851.0	2.76
100	13210	585.6	1854.4	3.16
140	13210	594.8	1847.9	3.1
180	13210	606.9	1832.7	3.01
300	13210	629.8	1851.0	2.93
400	13210	732.5	1852.9	2.52

Our final version of the marshaller uses the "just-in-time" idea with a threshold value of 100. We ask our last question: Does this version extract a significant penalty when marshalling *homogeneous* data? Table 7 shows the timings for this version of the marshaller, when marshalling collections of `Dummy` objects. This table is comparable to Table 1, and it shows that the JIT approach has almost no effect on performance for large homogeneous data sets.

Table 7. Performance when marshalling `Dummy` objects with threshold value of 100

Object Count	Bytes written	Jumbo + compilation	Kaffe OOS	Kaffe / Jumbo+compilation
1000	30000	37.0	151.7	4.09
10000	300000	149.9	1592.3	10.61
20000	600000	289.7	3167.4	10.93

It should be noted that if we have the opportunity to do off-line program generation, using specialized marshallers is the obvious decision, because we wouldn't have the run-time compilation cost. In this case, we would generate the specialized marshallers once before run-time and then at run-time we'd get the benefit of using them. Unfortunately off-line compilation is not always possible.

7 Sun's ObjectOutputStream

The aim of this paper is to show that RTPG using Jumbo is an easy and effective way to achieve higher performance. In this, we have reached the end of our

exposition. However, there are some loose ends to tie up. In particular, the reader may wonder how our code stacks up against the marshalling code that is delivered with HotSpot, which, as we have mentioned, uses unsafe, native code. (To be more specific, it uses the `sun.misc.Unsafe` class to access arbitrary memory addresses.) Another natural question is whether the kind of program generation we have done can be applied *to* the HotSpot code.

Table 8. Performance of Jumbo OOS vs. Sun OOS. Marshalling `Dummy` objects, program generation cost included, threshold value 100, incorporating lightweight hashtable

Object Count	Bytes written	Jumbo + compilation	Sun OOS	Jumbo+compilation / Sun
1000	30000	45.1	11.1	4.06
10000	300000	123.3	85.1	1.44
20000	600000	201.5	187.5	1.07

In table 8, we show the result of a test marshalling `Dummy` objects again, comparing Jumbo OOS (with threshold value of 100) to Sun OOS. To be fair to Jumbo OOS, we note that, in addition to using native methods, Sun OOS uses a custom, lightweight hashtable implementation, which is considerably more efficient than the standard implementation in this context. We incorporated this hashtable implementation into our code, too. In this test, Jumbo OOS is only 7% slower than Sun OOS on the largest data set, with 20,000 objects.

So, to summarize, while remaining entirely in the realm of verifiable Java code, we have obtained an implementation that can marshal large data sets nearly as fast as Sun's implementation.

Finally, we have experimented with applying RTPG to Sun OOS. We implemented Jumbo OOS and `ProgGen` using the same principles we discussed in Section 2 and 4, but based on Sun OOS instead of Kaffe OOS. (Although Sun OOS achieves its speed from using native methods in critical places, much of it is written in Java.) Comparing this version of Jumbo OOS to Sun OOS, we achieve speedups as high as 30% when run-time compilation cost is excluded. However, the crossover point is around 12,000 objects for homogeneous data sets.

8 Prior Work

Most work on optimizing marshalling is not directly comparable to ours in that the goal is not to optimize the existing, generic marshaller, but to create more efficient marshallers for special cases. For example, Nester et al. [1] require that classes that are to be marshalled must provide their own `writeObject` method, and also depart from the Sun serialization format in other ways which are valid in their environment, but not in general.

Manta [7] and Ibis [9] both use run-time code generation to produce specialized marshallers at run time. Their methods are different from ours: In Manta,

a compiler is invoked at run time (again requiring that all computers have a specified set-up in order to use their system); in Ibis, a specially built program generator producing JVM code has been written just to generate serializers.

Run-time program generation is the topic of many papers. The system closest to Jumbo is DynJava [16]. DynJava has certain restrictions, such as disallowing run-time generation of class names, which suggest that the translation from Kaffe OOS to DynJava might not be as straightforward as the translation to Jumbo; these restrictions are fundamental, as DynJava is type-safe. Nonetheless, we assume that, in general, DynJava could be used for marshalling much as we have done. Serialization is used as an example in two papers on RTPG systems that we know of. Neverov and Roe give the definition of a multi-stage language called Metaphor [17], in which, in principle, serialization code can be generated in a *type-safe* manner. However, they do not tackle the entire Java serialization specification, and it is not clear whether their techniques could scale to this case. Consel et al. [18] discuss marshalling for C, using the C-based Tempo system.

9 Conclusions

We have shown that marshalling code in Java can be highly optimized by generating marshallers at run-time. The speedup we obtained was an order of magnitude when compared to the marshalling code of Kaffe. For some data sets we nearly reached the speed of Sun's object serializer, which extensively uses unsafe native code, while staying entirely in the realm of verifiable byte code.

We applied a heuristic approach similar to just-in-time compilation to lower the break-even point for heterogeneous data sets. Another method which can further decrease runtime compilation cost is to optimize program generators statically. This approach is discussed in [19].

We have also shown that the transformation from the classical code to program generating code using `Jumbo` is straightforward. It does not require skills beyond ordinary programming. We conclude that considering this fact and the high speedup we obtained, optimizing marshalling is a potentially useful application of run-time program generation.

Jumbo itself, and all the code used in the experiments for this paper, can be obtained at `loome.cs.uiuc.edu/Jumbo/index.php`.

Acknowledgements

We would like to thank the (anonymous) reviewers, whose detailed comments on this paper were extremely valuable.

References

1. Nester, C., Philippsen, M., Haumacher, B.: A more efficient RMI for Java. In: Proc. of the ACM 1999 Java Grande, New York, NY, USA, ACM Press (1999) 152–159

2. Kamin, S., Clausen, L., Jarvis, A.: Jumbo: Run-time Code Generation for Java and its Applications. In: Proc. of the Intl. Symposium on Code Generation and Optimization (CGO '03), IEEE Computer Society (2003) 48–56
3. Clausen, L.: Optimizations In Distributed Run-time Compilation. PhD thesis, University of Illinois at Urbana-Champaign (2004)
4. Kamin, S., Callahan, M., Clausen, L.: Lightweight and Generative Components-2: Binary-Level Components. In: Intl. Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '00), Springer-Verlag (2000) 28–50
5. Kamin, S.: Routine Run-time Code Generation. In: Companion of the 18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), ACM Press (2003) 208–220
6. Kamin, S.: Program Generation Considered Easy. In: Proc. of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '04), ACM Press (2004) 68–79
7. Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H.E., Kielmann, T., Jacobs, C., Hofman, R.: Efficient Java RMI for Parallel Programming. ACM Trans. Program. Lang. Syst. **23** (2001) 747–775
8. Veldema, R., Philippsen, M.: Compiler Optimized Remote Method Invocation. In: IEEE International Conference on Cluster Computing (CLUSTER'03). (2003) 127
9. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R.F.H., Jacobs, C.J.H., Kielmann, T., Bal, H.E.: Ibis: A Flexible and Efficient Java-based Grid Programming Environment. Concurrency and Computation: Practice and Experience **17** (2005) 1079–1107
10. Kaffe JVM. (`http://www.kaffe.org`)
11. Java Object Serialization Specification. `http://java.sun.com/j2se/1.4.2/docs/guide/serialization/spec/serialTOC.html`
12. Taha, W., Sheard, T.: MetaML and Multi-stage Programming with Explicit Annotations. Theoretical Computer Science **248** (2000) 211–242
13. Java 1.4.2 API Documentation. (`http://java.sun.com/j2se/1.4.2/docs/api/`)
14. IBM-JVM. (`http://www-106.ibm.com/developerworks/java/jdk/`)
15. Advanced Programming for the Java 2 Platform: Ch. 8; Performance Features, Tools. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html`
16. Oiwa, Y., Masuhara, H., Yonezawa, A.: DynJava: Type Safe Dynamic Code Generation in Java. In: The 3rd JSSST Workshop on Programming and Programming Languages. (2001)
17. Neverov, G., Roe, P.: Metaphor: A Multi-stage, Object-Oriented Programming Language. In: Proc. of the Third Intl. Conf. on Generative Programming and Component Engineering (GPCE '04). (Lecture Notes in Computer Science)
18. Consel, C., Lawall, J.L., Meur, A.F.L.: A Tour of Tempo: A Program Specializer for the C Language. Sci. Comput. Program. **52** (2004) 341–370
19. Kamin, S., Aktemur, T.B., Morton, P.: Source-level optimization of run-time program generators. In: Generative Programming and Component Engineering (GPCE '05). (2005)