The Report Committee for Ryan William Harrod
Certifies that this is the approved version of the following report:

Compression of a Context-Based Marshalling Methodology in Java

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Sarfraz Khurshid

Thomas Graser

Compression of a Context-Based Marshalling Methodology in Java

by

Ryan William Harrod, B.S.C.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2016

Abstract

Compression of a Context-Based Marshalling Methodology in Java

Ryan William Harrod, M.S.E.

The University of Texas at Austin, 2016

Supervisor: Sarfraz Khurshid

This report focuses on a method for serializing messages in Java which is somewhere between a customized solution and Java's built-in serialization. Customized serialization will lead to the fastest, most optimized code, but will require more development time. Java's built in serialization, on the other hand, requires almost no effort to use but is slow, and uses more than is strictly necessary. This project gives a third option which requires a little more effort than Java's built in serialization, but still provides fast serialization speed, and low message sizes. The tool presented here can take in requirements for a message to transmit, which includes numbers, strings, booleans, and enumerations, and the output is a collection of Java classes which will serialize and deserialize that message. In turn, this generated library can be shared between projects within an organization, or even with third parties. The original message definition becomes a sort of interface control document, while the resulting code becomes a library which can be used to communicate with the defined messages, without requiring users of that message to develop their own code.

Table of Contents

List of Figures vi

Chapter 1: Previous Work 1

Motivation 1

Approach 2

BitQueue 4

Other Work 4

Results 5

Chapter 2: What's New 6

New Motivation 6

Chapter 3: Generated Code Improvement 8

Chapter 4: Junit Tests 10

Chapter 5: Enumerations 12

Chapter 6: Compression Results 15

Chapter 7: Conclusion 23

Appendix A 24

Glossary 25

References 27

List of Figures

Figure 1: Uncompressed Message Sizes (in Bytes) [6] 16

Figure 2: Compression Ratios.....17

Figure 3: Compressing Individual Messages.....18

Figure 4: Compressed Message sizes. 19

Figure 5: Size Comparison between uncompressed marshalled messages and compressed serialized messages. 19

Figure 6: Grouped Compression Results 20

Figure 7: Compressing Each Group Individually 21

Chapter 1: Previous Work

Previous work for this paper was completed in the Spring 2016 semester for Verification and Validation. This section aims to summarize the work as it is written in "Model, Test and Verify a Custom Message Encoding in Java" by Emil Bsaibes and Ryan Harrod [6].

MOTIVATION

It's common today to see applications passing data between clients and servers, between nodes of a distributed network, between back-end systems, etc. Some of the most popular formats include text-based formats such as XML and JSON. These formats have gained popularity due to their widespread support in major languages such as C++'s Boost libraries, Java's SDK, and .Net. A shortcoming of these algorithms is that they are using an inherently inefficient method of transmitting data by encoding it in string format. Consider the integer "1,234,567,890". This integer can be represented as a typical signed 32-bit integer in a total of 4 bytes. In XML, encoding this integer might look something like these examples:

<FieldName value="1234567890"/> (32 characters)

OR

<FieldName>1234567890</FieldName> (33 characters)

So by entrusting the encoding of the data to a simple and easy-to-use XML or JSON library, the size of data being sent can grow immensely.

Also considered in this paper is Java's built-in serialization capability. Simply by writing a class so that it "implements Serializable", a class in Java can be serialized to and from a byte array. This makes serializing data in Java easy, but there are some hidden costs

associated with Serializable. First, each Serializable object must have a 64-bit Long integer for as a unique identifier. This unique ID allows the receiving code to know which class is being serialized, so that it can be deserialized from a byte array properly. As this paper later shows, the act of serializing an object costs quite a bit of time compared to the encoding method developed herein.

Now, in most cases where this size and performance inefficiency start affecting the software being developed, the go-to solution is for the developers to implement a customized serialization. This method can take a considerable amount of development time, and while it might work for development environments that can spare the hours to research and develop a custom binary format, customization may not work so well for small teams that have tight time constraints.

The work in the first paper aimed at coming up with a custom encoding method which would try to act as a middle ground in terms of effort between customized serialization and using simple or easy-to-use libraries, such as those available for XML, JSON, or Java's built-in serialization. And, ideally, the custom encoding method would be more performant than the string encoding or Java's serialization. The results of the paper show that the custom encoding uses less space, and performs much faster than the alternatives.

APPROACH

For starters, we had to define a message format. The grammar for our messages is as follows:

Message -> (Group | List | Field)*

Group -> (Group | List | Field)*

List-> Group | Field

Field -> ValueType

ValueType -> String | Integer | Float | Boolean

Following this grammar, we wrote software which would marshall and unmarshall each message element. In most cases marshall/unmarshall is synonymous with serializing/deserializing, but for the purposes of this paper, marshalling will refer to our project's method of encoding, while serializing will refer to Java's built-in serialization.

Our project not only marshalls messages of the defined grammar, but also generates code that will do the marshalling. The purpose of generating code has many benefits: the generated code exists as its own entity, it can be placed anywhere in your source control management system, it can be modified by developers for fine-tuning purposes, and it can even be wrapped up in a JAR file and handed to third parties which allows them to effortlessly integrate your messages into their system (eg. If you're offering an API which transmits data over TCP/IP).

There are two primary techniques used by our software in order to minimize the size of the marshalled bytes. The first technique is to apply a context to each variable. For example, if you want to represent the day of the year as an integer, the naïve approach would be to use a 32-bit integer, whereas our software will calculate that it only needs 9-bits when marshalling. The second technique is the use of a 'presence indicator'. This is a bit which may or may not precede a message element in the marshalled bytes. If the field is deemed 'mandatory', then it must always be marshalled, and no presence indicator is required. If the field is 'optional', then the presence indicator will be '0' when the field is not present in the marshalled bytes, and it will be '1' when the field is present. To illustrate how this works, again consider a 4 byte integer. If the field is mandatory, we simply marshall all 4 bytes of the integer into the message:

0000 0000 0000 0000 0000 0000 0000 0000

If the field is optional, and is not present in the marshalled bytes, then we simply have the presence indicator equal ‘0’:

0

Otherwise, the presence indicator is ‘1’, followed by the field’s data:

1 0000 0000 0000 0000 0000 0000 0000 0000

The underlined bits above denote the presence indicator. The power behind the presence indicator can potentially lead to huge improvements in space efficiency. Consider two types of messages: one where all elements are optional, and one where all elements are mandatory. In the message where everything is mandatory, the user knows upfront exactly how many bytes each message will utilize – this can be useful when sending bytes of streams, and a sync pattern can be built into the message. In the message where everything is optional, you have the potential for sending messages which are only deltas of previously sent messages, and the minimum size, in bits, is simply the number of elements. You could imagine Internet of Things devices which use low power sending delta updates only as needed, and leaving the history/tracking up to the receiver. And, of course, everything between these two extremes is also possible.

BitQueue

The primary element of our project’s software is known as the BitQueue. This is a class which extends Java’s built-in BitSet. The idea of the BitQueue is that we should be able to push each element of the message onto this queue, and then convert the whole thing to an array of Bytes. The receiving ends does the same thing in reverse.

Other Work

In addition to the core aspects of our software, we developed a system of ‘RepOk’ functions [7] which would validate the structure and configuration of a message, prior to

generating code. This ensures quite a few things, but the most important is that each message element has a unique name. Since the generated code all ends up in the same package, conflicting names would mean one or more classes would be overwritten, and the generated code would be useless.

Additionally, we used the tool JPF (Java Path Finder) and its backtracking capabilities to generate Junit tests which tested the BitQueue code. This helped us with initial testing of the core functionality of our software. We also used JPF to help generate some test messages – these messages could then be turned into an XML string (but not return from an XML string back to a Java object, due to time constraints), serialized with Java's serialization, or marshalled using our code.

As a final note, we had intended to generate Junit tests which would test the generated marshalling code, but the only test that existed was to test the symmetry of marshalling and unmarshalling (the object you get out after unmarshalling should be equal to the object that went into marshalling).

RESULTS

The results were extremely satisfying. We compared XML, Java's Serialization, and the marshalling in terms of how many bytes are in the messages, as well as the time to generate the bytes. As expected, XML generated the largest messages, and was the slowest. The marshalling algorithm also outperformed Java's serialization. The marshalled message size was roughly $1/10^{th}$ of the serialized size, and the marshalling code ranged between 5-120 times faster than the serialization.

Chapter 2: What's New

New Motivation

It should be noted that the output of the marshalling algorithm is simply a byte array, so it can act as a payload in anything that transmits bytes. As such, the implemented software doesn't take any steps to implement the slew of services you might get by using UDP, TCP, or other transmission protocols. That means even simple things such as error checking are up to the user to handle.

This paper focuses on compressing the data. Some formats compress really well, with formats like BMPs and strings compressing more than 90% [5]. However other formats, such as video formats and ISO images, are already compressed and can't be further compressed [5]. We might expect the resulting byte arrays to still follow the same pattern – XML being the largest, Java's serialization coming in second, and the marshalled messages being smallest. But due to the variability of compression, this may not end up being the case.

Also included in the new functionality is the ability to marshall enumerations. This was originally intended to be one of the ValueTypes in the original work, but was left out due to time constraints. These enumerations aren't highly complex, but they do contain some neat features which will be explained later.

I also wanted to expand on the Junit tests which get generated for each message element. Some users will invariably have to provide code coverage for any source code they have, and the single unit test which was generated in the previous work does not provide full statement or branch coverage. Providing unit tests will also help users who end up having to modify the generated code. The generated code in that case will act as 'step one', and the user would fine tune it as necessary, while hopefully still not using as much

time as the custom serialization approach mentioned before. In this case, the unit tests give the user a baseline for what needs to work in order for the marshalling to operate properly.

Finally, I also looked into some of the improvements in the code that is automatically generated to marshall message elements. Some of the generated code from the original paper had unnecessary branches in them. While this would be unlikely to negatively impact run time due to the JIT compiler optimizing code [3], fixing these pieces was relatively low-hanging fruit.

Chapter 3: Generated Code Improvement

One of the Value types is the IntegralType. In code, an IntegralType can be represented as a Byte, Short, Integer, or Long, and can be marshalled into a field ranging from 1 to 64 bits.

Particularly, the generated code for setting an Integer field has been improved to conditionally check for the minimum and maximum bound if and only if these bounds are not the MINIMUM and MAXIMUM for the IntegralType type being marshalled (Byte, Short, Integer, Long). This particular change is mostly aesthetic. Here is an example of the previously generated code:

```
@Override
public final void setValue(Long val)
{
    if (val <= MAXIMUM && val >= MINIMUM)
    {
        value = val;
        setPresenceIndicator(PresenceIndicator.IS_PRESENT);
    }
    else
    {
        throw new IllegalArgumentException("INT_64 value is out
of range.");
    }
}
```

In the code above, the incoming value is always checked to be within the minimum and maximum, and this is true even if those bounds are the type's absolute minimum and maximum (in this case Long.MIN_VALUE and Long.MAX_VALUE). The branch prediction in Java would very quickly realize that the 'else' statement never gets hit and simply always 'predict' the if-statement's body will execute. So improving this statement was simply a matter of taste and coding style, and not necessarily a matter of performance. Now, the code will appropriately include min and max checks only as needed, so it's

possible to have both checks, no checks, checking the min value only, or checking the max value only.

Similar to the previous issue, there was an unnecessary branch being created when marshalling a “MANDATORY” field. The previous code looked like this:

```
@Override
 public final void marshall(BitQueue bits)
 {
        marshallPresenceIndicator(bits);
        if (isPresent())
       {
              bits.append(getValue(), FIELD_SIZE);
       }
 }
```

In the above code, “marshallPresenceIndicator” will place either a 0 or 1 onto the BitQueue if the element’s presence indicator is “IS_NOT_PRESENT” or “IS_PRESENT”, respectively. It will not place any bit on the BitQueue if the presence indicator is “MANDATORY”. The ‘if (IsPresent())’ statement checks to see if the message element is either “IS_PRESENT” or “MANDATORY”, but the check to see if the element is “MANDATORY” isn’t needed: for elements that are required to be in the message, that particular if statement will never be false. So one improvement that has been made in generating the code is to leave out this if-statement and the marshallPresenceIndicator call when the element is mandatory.

Chapter 4: Junit Tests

As mentioned before, one of the goals of this project is to generate more unit tests, so that the generated code has full statement and branch coverage. The original unit test that was generated did a very simple thing: it took a message element, marshalled it, unmarshalled it into a new object, and compared both objects for equality. This symmetry is absolutely necessary – the object that goes in must be the object that comes out, otherwise data has been misinterpreted or lost somewhere. This is by far the most important test in terms of functionality, but it starts from an object that is 'safe'.

Each generated class has a 'mock' method which mocks values for primitive types (int, long, string, etc), as well as calls its child-objects' mock methods. Each mock method is 'safe' in that it's generated in such a way that a bad value isn't introduced anywhere. Eg. If an integer must be between 0 and 365, then the mock method never tries to set the integer to 1029383. With this in mind, some of the new Junit tests start from scratch – either without mocked objects, or they use a mocked object and try to violate constraints.

The obvious tests for IntegralTypes and FloatTypes are to check for their min and max value constraints. For all generated types, an IllegalArgumentException will be thrown if an attempt to set its value is outside of its min and max values. Junit has a built-in feature which allows a unit test to run code which is expected to throw an exception. Here's an example of one of the generated unit tests:

```
/**
 * Tests setting a bad value below the minimum.
 */
@Test(expected = IllegalArgumentException.class)
public void testSetBelowMin()
{
    final Minute testSubject = new Minute();
    //CHECKSTYLE:OFF Magic numbers
    testSubject.setValue(-1);
    //CHECKSTYLE:ON Magic numbers
}
/**
 * Tests setting a bad value above the maximum.
 */
@Test(expected = IllegalArgumentException.class)
public void testSetAboveMax()
{
    final Minute testSubject = new Minute();
    //CHECKSTYLE:OFF Magic numbers
    testSubject.setValue(60);
    //CHECKSTYLE:ON Magic numbers
}
```

These two tests have been generated to test a message element which represents a "Minute". Minutes can range from 0-59, so both -1 and 60 are outside of the allowed range. As expected, when these tests run an exception is thrown, but in this case that exception satisfies the conditions of a successful test.

Chapter 5: Enumerations

The enumerations which can be generated and used are very straightforward, for the most part. Two key features of these enumerations are: the ability to assign an integer to members of an enumeration, and the ability to assign a default value. Each member of the enumeration must, at a bare minimum, have its own unique name. This is one of the constraints that is checked in the "RepOk" function prior to generating any code. If there happen to be more than one member of the enumeration with the same name, no code will be generated.

The first important feature of these enumerations is the ability to assign an integer value to specific members. We can do this by making a private constructor in the enumeration which takes an integer as a parameter, and stores that integer as a member variable of the member. For example:

```
public enum EPowerLevel
{
    //CHECKSTYLE:OFF
    Low(0),
    Medium(1),
    High(2);
    //CHECKSTYLE:ON

    /**
     * The assigned value.
     */
    private int value;

    /**
     * @param val The assigned value.
     */
    EPowerLevel(int val)
    {
        value = val;
    }
    …
}
```

In the enumeration above, the integers which are assigned to each enumeration member happen to be consecutive numbers, 0-2, but that is not a requirement. The numbers

could have gaps; the only requirement is that they must be unique, just like the member names. The reason for this is because the assigned integer value will be the one that gets marshalled, and any code that received a marshalled enumeration that had ambiguous integer assignments would not be able to properly reconstruct the message in the remote code. Unique integer assignments are another aspect of the enumeration which is checked in the RepOk function prior to code generation. In addition, if any of the enumeration members have an assigned integer value, then they must all have an assigned integer value.

The other key feature of an enumeration is the ability to assign a default value. This comes into play when trying to find a member of the enumeration based on its assigned integer. In the case where a lookup is performed using an integer which is not assigned anywhere in the enumeration, the returned value will either be the first member of the enumeration, or the default. Consider the following enumeration members:

```
public enum EShapeType
{
    Point(0),
    Line(1),
    Triangle(2),
    …
    Unknown(1011),
    Spare(1012),
    Reserved(1013),
    Circle(1023);
}
```

Three of the values – “Unknown”, “Spare”, “Reserved” – are usually seen in a document which is intended to be a living document, with intentionally unused integers so that they can be re-purposed in the future without having to redefine the enumeration. In the above case, we could make a case for several of the members to be the default – or to even just allow the first member to act as the default. The RepOk function will check to make sure the default value is unique prior to code generation; there cannot be more than one default value.

However, neither of those two features are required in order to marshall an enumeration. In the event that there are no integer assignments and no default value, the default value becomes the first enumeration member, and the integers which are marshalled start at 0 and increment by 1 as you go down the list of enumeration members. This process is symmetric and the remote code is guaranteed to unmarshall the correct enumeration member.

Chapter 6: Compression Results

Obtaining benchmark results in Java can be tricky [3]. In order to increase confidence in the execution times of compressing each of the three types of data (XML string, Java serialized object, and a marshalled object), several techniques needed to be employed. As Brent Boyer mentions in his article [3], Java's JIT compiler can throw a wrench into benchmarking by optimizing out dead code, or even learning which code blocks are more likely to run. Unfortunately there's nothing I could do to prevent the dead code issue, because the Deflater code was written by a third party. However, to "warm-up" the JIT compiler I ran all of the data through the Deflater, prior to taking any time measurements. Once that's completed, I tested each individual set of data at a time so that their average execution times could be compared.

In total, there were 144 test messages. These test messages were programmatically generated using JPF, and included all types of message elements. Each test message was instantiated as an object, and that object was converted to XML, serialized with Java's built-in serialization, and marshalled. The byte array for each of these was kept for later testing the compression algorithm. With all of the byte arrays generated, I then measured how many times the Deflate [2] algorithm could compress the bytes of each message in a time period of 10 seconds. The Deflate algorithm is a popular compression algorithm which is used as part of the .ZIP file format specification [1] and can even be found in the HTTP protocol [4]. Since there are a wide variety of compression algorithms available, I chose to use the deflate algorithm because it would be the most likely candidate for the target audience: one that wants to improve the efficiency of their software, but may not necessarily have the ability to find the absolutely most efficient compression algorithm for the data they need to transmit.

As a side effect of this methodology, and as Boyer mentions in his article, the JIT compiler could possibly 'train' itself during execution to become better at the particular type of compression it's executing. For example, it could analyze the code the Deflater uses while compressing XML strings, and optimize how the code executes, despite the original warm up. To give an idea of the relative byte sizes, here's a chart showing the uncompressed sizes of each message:

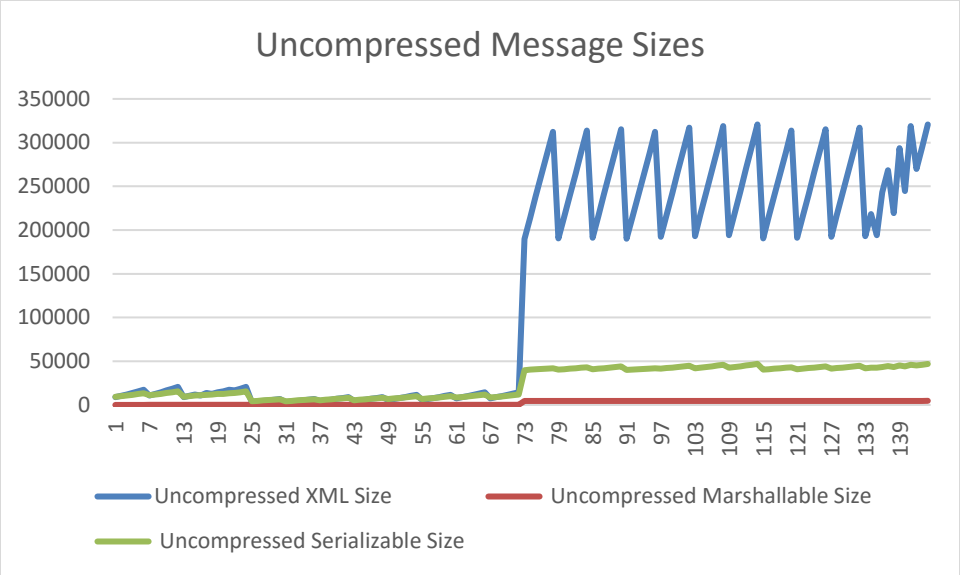

Figure 1: Uncompressed Message Sizes (in Bytes) [6]

The data for the above chart is in the original work done for the Verification and Validation course, and shows that the marshalled messages are always smaller than the serialized and XML message sizes. The messages are ordered from left to right based on the size of the marshalled message, with the smallest message being message 1, and the largest being message 144. Also for comparison, here are the relative compression ratios:

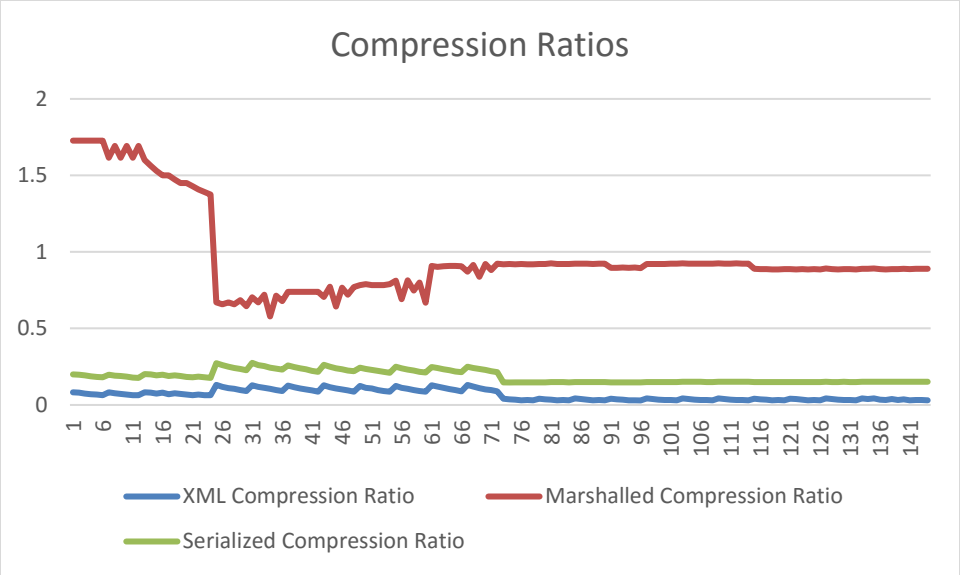

Figure 2: Compression Ratios

Like the previous chart, the messages in this chart are ordered based on the uncompressed size of the marshalled message, with the smallest messages being first. The above chart shows relative compression ratios. We see a similar trend for both serialized and XML data, with the compression ratio being more than 50%. The marshalled data, on the other hand, seems to struggle and compresses very little in comparison, and with messages that were too small, the compressed message ended up being 1.5-2.0 times larger than the original. It would be worth investigating even larger messages in the future, to see if compression remains close to 0% as shown above, or if large messages are able to meet the compression ratios shown by XML and serialized bytes.

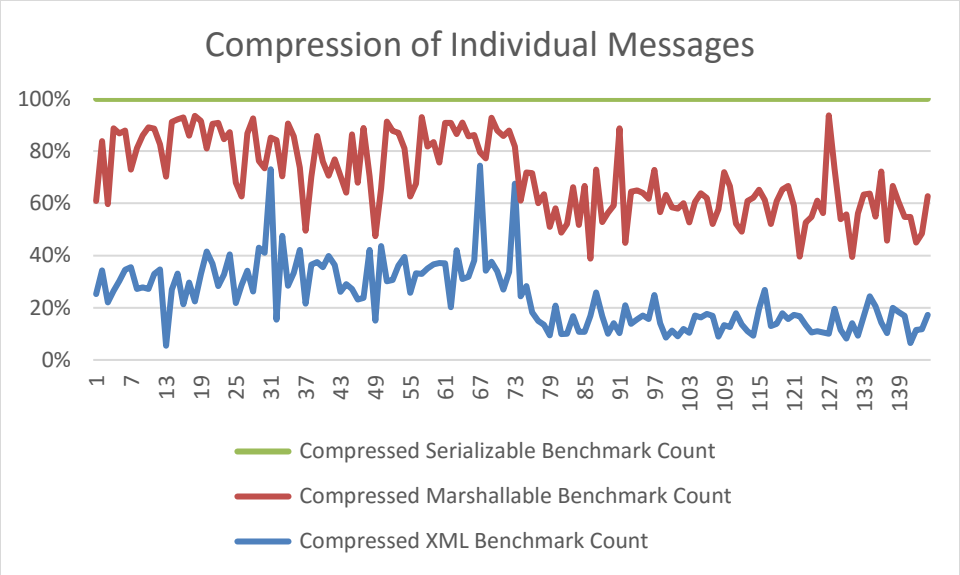

Figure 3: Compressing Individual Messages

The chart above shows how many times an individual message could be compressed in 10 seconds. In all cases, the serialized message was compressed the most times, so the other data is shown in relation to how many times the serialized message was compressed. This result is particularly interesting, because it suggests that the deflate algorithm has a preference, in terms of speed, towards the serialization that is built in to Java. The marshalled messages are significantly smaller than the serialized messages, but take longer to compress. However, it is noteworthy that some of the marshalled messages are so small in size, the compressed versions ended up being larger than the original. The smallest uncompressed marshalled message was a mere 11 bytes in size. Compression didn't improve the size of the marshalled messages until the uncompressed data was roughly 70 bytes in size.

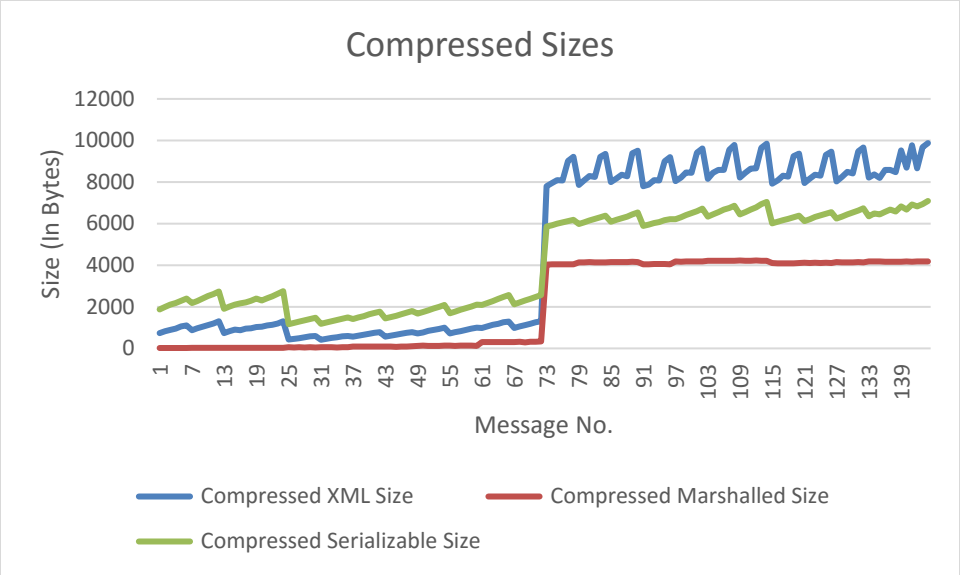

Figure 4: Compressed Message sizes.

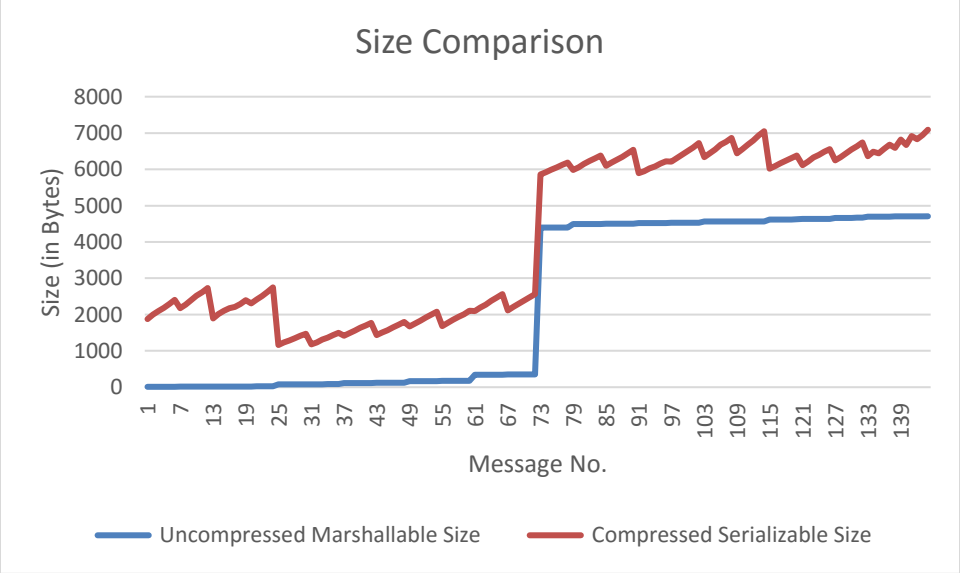

Figure 5: Size Comparison between uncompressed marshalled messages and compressed serialized messages.

The messages in the above chart are sorted based on the size of the original (uncompressed) marshalled data. One important note is that there are effectively two types of data in the messages: due to how the messages were generated, half of the messages are

small in size, while the other half are much larger. The above chart shows that the compressed marshalled data is roughly 2-2.5 times smaller than the XML data, and about 2/3 the size of the compressed serialized data.

In the chart below, instead of compressing each individual message one at a time, I compressed all similarly generated byte arrays together over a period of ten seconds. Eg. I took the 144 byte arrays associated with XML strings and compressed them all as many times as possible over 10 seconds. In this iteration, I compressed the XML messages as many times as possible in 10 seconds, then compressed marshalled messages, then compressed the serialized messages, and then repeated this whole process 50 times to see how the average behavior looks.

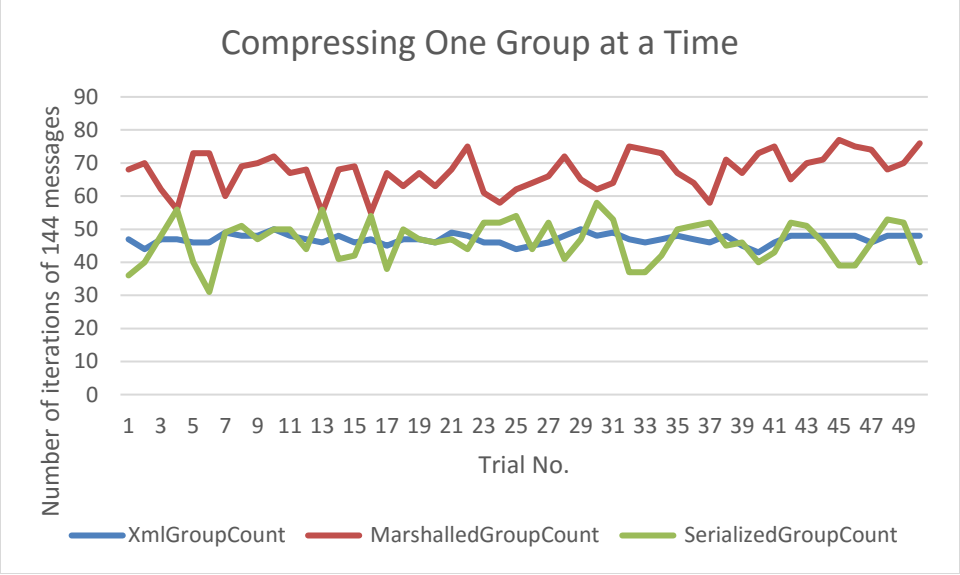

Figure 6: Grouped Compression Results

As you can see in the above chart, the number of marshalled messages that can be compressed beats the number of XML or serialized messages. Oddly enough, the XML messages and serialized messages are, on average, roughly equal in terms of performance

time. Due to the jagged nature of the marshalled and serialized data in this test, I was interested to see if the JIT compiler was somehow optimizing itself between runs, and whether that had an effect on the outcome. Remember, in the above chart, the XML messages were tested, then the marshalled, then the serial. In the next test, I ran all 50 iterations of the XML messages, then 50 iterations of the marshalled messages, and then 50 iterations of the serialized messages.

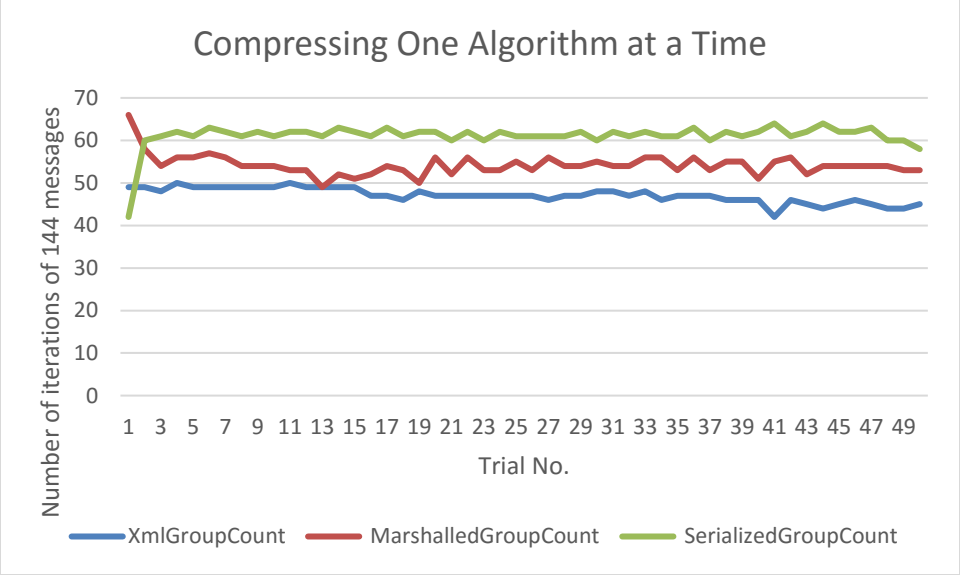

Figure 7: Compressing Each Group Individually

Here we see a clearer result, and this execution probably represents what we would see in the real world. In a normal application, we likely wouldn't switch between three methods of generating messages, so the JIT compiler would be free to optimize towards one message type. There is a bit of a twist with this data, though. In each case, we can see the JIT compiler uses the first messages to train itself, and then we have rather flat performance for the rest of the messages. Interestingly, it's the serialized messages which perform the best here, so more serialized messages were compressed within the 10 second time period than the others. XML is still the worst, though this is probably due to its much

larger byte size. Marshalled messages were in the middle, despite being the smallest in overall byte size.

Chapter 7: Conclusion

There was one odd result when looking at the performance of compressing marshalled messages. We would expect the JIT compiler to optimize well when running the compression algorithm on only one set of data. The results instead show that compressing marshalled data is faster when periodically running the other sets of data through the Deflater. When running all three sets in a round-robin fashion, the marshalled data was able to get through ~65 iterations in 10 seconds. But when running just the marshalled data through the Deflater, it was only able to execute ~55 iterations in 10 seconds. XML messages are roughly the same in performance with less variance, and the serialized messages improve in performance, going from ~45 iterations in 10 seconds to ~60. This suggests that, either through design or luck, that the Java serialized data works well with the Deflate algorithm. Meanwhile, the marshalled data does the exact opposite. While the marshalled data is still typically smaller when it's compressed, it doesn't compress very much and it takes longer for the compression algorithm to run when compared to running on serialized data.

Based on these results, it would seem that at least for marshalled messages under ~50 KB, compression doesn't offer much. However, it's important to remember that the marshalled messages in their uncompressed form were still smaller than the serialized messages after compression. Ultimately, the marshalling method that has been developed outperforms Java's built in serialization. The usability of the software is still undecided because it has not been publicly released prior to this paper, but it should at least be easier than creating an entirely new and custom set of code for serialization.

Appendix A

The code used in this project has been posted on GitHub using the MIT license here: https://github.com/rwharrod/CBM

Perhaps most importantly, the code used to run the tests in this paper can be found in the repository mentioned above. The work related to compression can be found under the com.encoding.testJava package, including the code which warms up the JIT compiler, as well as all of the code used to benchmark the compression of XML strings, serialized data, and marshalled data.

Glossary

BooleanType – A ValueType which, once marshalled, is a single bit.

Deserialize – The process of undoing the serialize operation, going from a byte array to an object.

Encoding – The bit-level representation of data in a message or in storage.

EnumType – A ValueType which represents a Java ‘enum’ for marshalling.

FloatType – A ValueType which represents either a 32-bit ‘float’ or a 64-bit ‘float’.

IntegralType – A ValueType which can represent a Byte, Short, Integer, or Long object. An IntegralType’s size in bits can range from 1-63 in the current implementation.

JSON – JavaScript Object Notation

Marshall – The process of taking an object and encoding it to an array of bytes. This is usually synonymous with Serialize, but in the context of this paper, all references to ‘Marshalling’ refer to the code which has been developed.

Message Element – An element of a message which can be marshalled using the technique used in this paper.

Presence Indicator – Indicates whether a message element is present or not in the marshalled bytes. A presence indicator of ‘1’ means the message element follows after the presence indicator in the marshalled bytes. Message elements without a presence indicator are considered mandatory.

Recurrence Indicator – Similar to a presence indicator, a recurrence indicator is used while marshalling lists. A recurrence indicator of ‘1’ means another element of the list follows after the recurrence indicator in the marshalled bytes. A list is terminated by reaching a maximum allowed number of recurrences, or by a recurrence indicator of ‘0’, whichever comes first.

Serialize – The process of taking an object and encoding it to an array of bytes. This is usually synonymous with Marshall, but in the context of this paper, all references to 'Serialize' refer to Java's built-in serialization capability.

StringType – A ValueType which represents a string for marshalling.

Unmarshall – The process of undoing the marshall operation, going from a byte array to an object.

XML – Extensible Markup Language

References

[1] PKWARE, Inc., "APPNOTE.TXT - .ZIP File Format Specification" https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT [Online; accessed October 11, 2016]

[2] L. Peter Deutsch, "DEFLATE Compressed Data Format Specification version 1.3" https://tools.ietf.org/html/rfc1951#section-Abstract, May 1996[Online; accessed 11 October, 2016]

[3] Brent Boyer (24 June 2008), "Robust Java benchmarking, Part 1: Issues" http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html [Online; accessed 11 October, 2016]

[4] David Saloman, *Data Compression: The Complete Reference,* 3rd Edition, Springer-Verlag New York, Inc., 2004, Section 3.23.

[5] "Why don't some files compress very much?" http://kb.winzip.com/kb/entry/104/ [Online; accessed October 20, 2016]

[6] Emil Bsaibes, Ryan Harrod "Model, Test and Verify a Custom Message Encoding in Java", May 2016

[7] Barbara Liskov, John Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design,* Addison-Wesley, 2000.