Skyway: Connecting Managed Heaps in Distributed Big Data Systems

Khanh Nguyen, Lu Fang, Christian Navasca, Harry Xu, Brian Demsky

Shan Lu
University of Chicago, Irvine

University of California, Irvine
BIG DATA
BIG DATA
MR, Spark Apps
The managed runtime is costly
The managed runtime is costly
The managed runtime is costly
Send & Receive Objects

Scala

Data Shuffling

The managed runtime is costly
Skyway
Send & Receive Objects

Scala

Data Shuffling

The managed runtime is costly
OutputStream out = Shuffler.GetOutputStream(receiver_id);

for (Object o: outDataset) {
    out.writeObject(o);
}
OutputStream out = Shuffler.GetOutputStream(receiver_id);
for (Object o: outDataset) {
    out.writeObject(o);
}
InputStream in =
    Shuffler.GetInputStream(sender_id);

while (in.hasData()) {
    Object o = in.readObject();
    inDataset.store(o)
}

// deserialization

OutputStream out =
    Shuffler.GetOutputStream(receiver_id);

for (Object o: outDataset) {
    out.writeObject(o);
}
InputStream in =
    Shuffler.GetInputStream(sender_id);

while (in.hasData()) {
    Object o = in.readObject();
    inDataset.store(o)
}

**deserialization**

**serialization**

OutputStream out =
    Shuffler.GetOutputStream(receiver_id);

for (Object o: outDataset) {
    out.writeObject(o);
}
**Data transfer costs**

TriangleCounting over LiveJournal on Spark 2.1.0 with 3 slaves
Data transfer costs

TriangleCounting over LiveJournal on Spark 2.1.0 with 3 slaves
Data transfer

Sender

Receiver

Object
Data transfer

Sender

Receiver

Object

Serialization
Data transfer

Sender

Object

Serialization

Reflection. getField

Receiver
Data transfer

Sender

Receiver

Object

Serialization

Reflection.
getField
Data transfer

Sender

Object

Serialization

Reflection.
getField

Receiver
Data transfer

Sender

Object

Serialization

Reflection.
getField

Receiver

Binary

1001001100011011100
1011111101100011110
1101010010111010011
110111111111010001
Data transfer

Sender

Receiver

Object

Serialization

Reflection. getField

Binary

Network
Data transfer

Sender

Receiver

Object

Binary

Network
Data transfer

Sender

Receiver

Object

Binary

Deserialization

1001001100011011100
101111101100011110
1101010010111010011
110111111111010001
Data transfer

Sender

Receiver

Deserialization

Reflection.
allocate
Reflection.
setField

Reflection.

Binary

Object
Data transfer

Sender

Receiver

Object

Binary

Deserialization

Reflection.
allocate
Reflection.
setField

100100110001110111100011101111110110000010101000101001110001100100010110001011001101011111111101110
Data transfer

Sender

Object

Binary

Receiver

Reflection.
allocate
Reflection.
setField

Deserialization

6
Data transfer

Sender

Receiver

Object

Reflection. allocate

Reflection. getField

Reflection. setField

Serialization

Network

Deserialization

Binary
Data transfer

WANTED: a system-level solution
an analogy
an analogy
an analogy
an analogy
an analogy
an analogy
Our solution
Our solution

Sender

Receiver
Our solution

Sender

Object

Receiver
Our solution

Skyway
Our solution

Skyway

Sender

Object

Reflection.
allocate

Receiver

Reflection.
getField

Reflection.
setField
Our solution

Skyway

Sender

Receiver

Object

Reflection.
allocate

Reflection.
getField

Reflection.
setField
Our solution

Skyway

Overlapping computation and data transfer
Skyway Overview
Skyway Overview

– Implemented in OpenJDK 8
  • Modified the class loader, the object/heap layout, the Parallel Scavenge GC

– Efficiently handle data transfer:
  • Outperforms 90 serializers
  • Improves Spark by 36% (Java) - 16% (Kryo)
  • Improves Flink by 19%
Challenges
Challenges

1. Type representation
Challenges

1. Type representation
   - Automated global type numbering
Challenges

1. Type representation
   - Automated global type numbering
2. Pointer representation
Challenges

1. Type representation
   - Automated global type numbering

2. Pointer representation
   - Use relative offsets
Challenges

1. Type representation
   - Automated global type numbering
2. Pointer representation
   - Use relative offsets
3. Local JVM adaptation
Challenges

1. Type representation
   - Automated global type numbering
2. Pointer representation
   - Use relative offsets
3. Local JVM adaptation
   - Visible for garbage collection
Challenges

1. Type representation
   - Automated global type numbering
2. Pointer representation
   - Use relative offsets
3. Local JVM adaptation
   - Visible for garbage collection
4. Work pipelining
Challenges

1. Type representation
   - Automated global type numbering
2. Pointer representation
   - Use relative offsets
3. Local JVM adaptation
   - Visible for garbage collection
4. Work pipelining
   - Buffering
Type registries

Type Registry

Worker A

Type Registry A

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
<th>Metadata Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;java.lang.Object&quot;</td>
<td>1</td>
<td>klass for “java.lang.Object”</td>
</tr>
<tr>
<td>&quot;org.apache.spark.rdd.RDD&quot;</td>
<td>2</td>
<td>klass for “org.apache.spark.rdd.RDD”</td>
</tr>
<tr>
<td>&quot;java.lang.String&quot;</td>
<td>5</td>
<td>klass for “java.lang.String”</td>
</tr>
</tbody>
</table>

Worker B

Type Registry B

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
<th>Metadata Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;java.lang.Object&quot;</td>
<td>1</td>
<td>klass for “java.lang.Object”</td>
</tr>
<tr>
<td>&quot;java.lang.String&quot;</td>
<td>5</td>
<td>klass for “java.lang.String”</td>
</tr>
<tr>
<td>&quot;org.apache.spark.scheduler.Task&quot;</td>
<td>120</td>
<td>klass for “org.apache.spark.scheduler.Task”</td>
</tr>
</tbody>
</table>

Cluster
Output & Input buffer
Output & Input buffer

Output buffer
Output & Input buffer
Output & Input buffer

• Segregated by *receivers*
• *One* for each receiver
Output & Input buffer

- Segregated by **receivers**
- **One** for each receiver
- In **native, off-the-heap** memory
Output & Input buffer

Input buffer

Output buffer

• Segregated by *receivers*
• *One* for each receiver
• In *native, off-the-heap* memory
Output & Input buffer

- Segregated by **receivers**
- **One** for each receiver
- In **native, off-the-heap** memory
Output & Input buffer

Input buffer

- Segregated by **senders**
- **Multiple** for each sender

Output buffer

- Segregated by **receivers**
- **One** for each receiver
- In **native, off-the-heap** memory
Output & Input buffer

**Input buffer**

- Segregated by **senders**
- **Multiple** for each sender
- In managed **heap**

**Output buffer**

- Segregated by **receivers**
- **One** for each receiver
- In **native, off-the-heap** memory
Example: Serialization

Integer[]

3 0xaa 0xbb 0xcc

Integer 10

Integer 20

Integer 30
Example: Serialization

writeObject()

Integer[] 3 0xaa 0xbb 0xcc

Integer 10

Integer 20

Integer 30
Example: Serialization

writeObject()
Example: Serialization

`writeObject()`

Output buffer in native memory
Example: Serialization

`writeObject()`

```
Integer[] 3 0xaa 0xbb 0xcc
```

```
Integer 10
Integer 20
Integer 30
```

Output buffer in native memory

Offset

0
Example: Serialization

writeObject()

Integer[] 3 0xaa 0xbb 0xcc

Integer 10

Integer 20

Integer 30

Output buffer in native memory

Offset

0
Example: Serialization

writeObject()

Type Registry

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;java.lang.Integer&quot;</td>
<td>6</td>
</tr>
<tr>
<td>&quot;[java.lang.Integer&quot;</td>
<td>7</td>
</tr>
</tbody>
</table>

Output buffer in native memory

Offset

0
Example: Serialization

writeObject()

Output buffer in native memory

Type Registry

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;java.lang.Integer&quot;</td>
<td>6</td>
</tr>
<tr>
<td>&quot;[java.lang.Integer&quot;</td>
<td>7</td>
</tr>
</tbody>
</table>
Example: Serialization

`writeObject()`

Output buffer in native memory

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>“java.lang.Integer”</td>
<td>6</td>
</tr>
<tr>
<td>“[java.lang.Integer”</td>
<td>7</td>
</tr>
</tbody>
</table>
Example: Serialization

```java
writeObject()
```

Output buffer in native memory

Type Registry

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;java.lang.Integer&quot;</td>
<td>6</td>
</tr>
<tr>
<td>&quot;[java.lang.Integer&quot;</td>
<td>7</td>
</tr>
</tbody>
</table>
Example: Serialization

writeObject()

Output buffer in native memory

Type Registry

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>“java.lang.Integer”</td>
<td>6</td>
</tr>
<tr>
<td>“[java.lang.Integer”</td>
<td>7</td>
</tr>
</tbody>
</table>

Offset

0 7
Example: Serialization

```java
writeObject()
```

![Serialization Diagram]

Type Registry

<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>“java.lang.Integer”</td>
<td>6</td>
</tr>
<tr>
<td>“[java.lang.Integer”</td>
<td>7</td>
</tr>
</tbody>
</table>
Example: Serialization

writeObject()
Example: Serialization

```plaintext
writeObject()
```

**Output buffer in native memory**

```
7 3 7 11 15 6 10 6 20 6 30
```

**Type Registry**

```
<table>
<thead>
<tr>
<th>TypeString</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;java.lang.Integer&quot;</td>
<td>6</td>
</tr>
<tr>
<td>&quot;[java.lang.Integer&quot;</td>
<td>7</td>
</tr>
</tbody>
</table>
```

```
Integer[] 3 0x0 0xbb 0xcc
Integer 10
Integer 20
Integer 30
```

```
Offset
0 7 11 15
```
Example: Deserialization

Offset 0 7 11 15

7 3 7 11 15 6 10 6 20 6 30
Example: Deserialization

Offset 0 7 11 15

readObject()
Example: Deserialization

Input buffer in heap

readObject()
Example: Deserialization

readObject()

Input buffer in heap

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>7</th>
<th>11</th>
<th>15</th>
</tr>
</thead>
</table>

Type Registry

<table>
<thead>
<tr>
<th>ID</th>
<th>MetadataObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>7</td>
<td>java.lang.Integer[]</td>
</tr>
</tbody>
</table>
Example: Deserialization

```
Example: Deserialization
```

```
Input buffer in heap
```

```
Integer[] 3 7 11 15 6 10 6 20 6 30
```

```
Type Registry
```

```
ID | MetadataObject
---|----------------
6  | java.lang.Integer
7  | java.lang.Integer[]
```
Example: Deserialization

```java
readObject()
```

Input buffer in heap

Type Registry

<table>
<thead>
<tr>
<th>ID</th>
<th>MetadataObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>7</td>
<td>java.lang.Integer[]</td>
</tr>
</tbody>
</table>
Example: Deserialization

```java
readObject()
```

Input buffer in heap

Type Registry

<table>
<thead>
<tr>
<th>ID</th>
<th>MetadataObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>7</td>
<td>java.lang.Integer[]</td>
</tr>
</tbody>
</table>
Example: Deserialization

```
Offset
0  7  11  15

readObject()
```

Input buffer in heap

```
[3 0xfb 0xff 0xf7 6 10 6 20 6 30]
```

<table>
<thead>
<tr>
<th>ID</th>
<th>MetadataObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>7</td>
<td>java.lang.Integer[]</td>
</tr>
</tbody>
</table>
Example: Deserialization

```
readObject()
```

Input buffer in heap

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>7</th>
<th>11</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer[]</td>
<td>3</td>
<td>0xfb</td>
<td>0x7</td>
<td>0xff</td>
</tr>
<tr>
<td>Integer</td>
<td>10</td>
<td>6</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>
```

Type Registry

<table>
<thead>
<tr>
<th>ID</th>
<th>MetadataObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>7</td>
<td>java.lang.Integer[]</td>
</tr>
</tbody>
</table>
Example: Deserialization

readObject()
Example: Deserialization

readObject()

Input buffer in heap

<table>
<thead>
<tr>
<th>ID</th>
<th>MetadataObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>7</td>
<td>java.lang.Integer[]</td>
</tr>
</tbody>
</table>
In the paper

- Cyclic references
- Shared objects
- Support for threads
- Interaction with GC
- Integrating Skyway in real systems
Evaluations - Microbenchmark

- Java Serializer Benchmark Set
  - Extensive performance evaluation with existing 90 serializers
**SKYWAY**

**GOOGLE’s Protobuf**

**Kryo (rec. by Spark)**
Evaluations – Real Systems

• Flink 1.3.2
  – 5 query answering applications
  – TPC-H datasets
Evaluations – Real Systems

- **Flink 1.3.2**
  - 5 query answering applications
  - TPC-H datasets
Evaluations – Real Systems

- **Flink 1.3.2**
  - 5 query answering applications
  - TPC-H datasets

On average, reduces end-to-end time by **19%**
Improvement Summary: Flink

Normalized Performance to built-in serializer

Ser. Time  Deser. Time  Execution Time
Improvement Summary: Flink

-23%
Improvement Summary: Flink

Normalized Performance to built-in serializer

-23%  -25%
Improvement Summary: Flink

-23%  -25%  -19%
Evaluations – Real Systems

• Spark 2.1.0
  – 4 applications: WordCount, PageRank, ConnectedComponents, and TriangleCounting
  – 4 datasets:
    LiveJournal, Orkut, UK-2005, and Twitter
Evaluations – Real Systems

• Spark 2.1.0
  – 4 applications: WordCount, PageRank, ConnectedComponents, and TriangleCounting
  – 4 datasets:
    LiveJournal, Orkut, UK-2005, and Twitter

On average, reduces end-to-end time
  by 16% (w.r.t. Kryo)
  by 36% (w.r.t. Java serializer)
Improvement Summary: Spark

Normalized Performance to Java Serializer

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improvement Summary: Spark vs. Java

Normalized Performance to Java Serializer

<table>
<thead>
<tr>
<th>Kryo Serial. Time</th>
<th>Skyway Serial. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kryo Deser. Time</td>
<td>Skyway Deser. Time</td>
</tr>
<tr>
<td>Kryo Execution Time</td>
<td>Skyway Execution Time</td>
</tr>
</tbody>
</table>

Execution Time
Improvement Summary: Spark

-38% vs. Java

Normalized Performance to Java Serializer

<table>
<thead>
<tr>
<th>Kryo Ser. Time</th>
<th>Skyway Ser. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kryo Deser. Time</th>
<th>Skyway Deser. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kryo Execution Time</th>
<th>Skyway Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improvement Summary: Spark

-38% vs. Java

Normalized Performance to Java Serializer

Kryo Ser. Time

Skyway

Kryo Deser. Time

Skyway

Kryo Execution Time

Skyway
Improvement Summary: Spark

-38% vs. Java

Normalized Performance to Java Serializer

-84%

Kryo Skyway
Kryo Skyway
Kryo Skyway

Ser. Time Deser. Time Execution Time
Improvement Summary: Spark

Normalized Performance to Java Serializer

Kryo vs. Kryo

- 0.05% improvement
- 16% decrease
- 38% decrease

Kryo Ser. Time

Kryo Deser. Time

Kryo Execution Time

Skyway

Skyway
Conclusion

• **Goal:** Reduce data transfer costs in Big Data systems

• **Solution:** **Skyway**, the first JVM-based serializer
  – Efficiently transfer data
  – Easy to integrate
Thank You!