# DJ
## Distributed JIT

Matthew Francis-Landau

UC Berkeley

September, 2015

# Structure of DJ

- Runtime
  - Performs dynamic code rewriting
  - Remote memory access
  - Distributed locks
  - **Ensure correctness of program regardless of distributed state
- JIT
  - High level placement/scheduling decisions about a program
  - Regardless of placement decisions program will continue to execute correctly
  - Could potentially be replaced with a user supplied JIT

# Some rewriting details

- Reads and writes of fields on an object are replaced with a inlinable method call
  - Replacement only happens if there is at least one instance of the object that is distributed
- Method calls when transformed to RPC, have a some bytecode inserted at the beginning to check if it should be an RPC call
- Array classes are replace so that all reads and writes can be observed and accessed remotely

# Example field access

```
// before
a = 5;
// rewritten
write_variable_a(this, 5);
// ...
static void write_variable_a(ObjectType self, int val) {
  if((self.__dj_class_mode & 0x2) != 0) {
    // .. redirect write
  }
  self.a = val;
}
```

- `write_variable_a` is a static method call which means that the JVM can inline it
- This rewrite takes place one there exists an instance of the class that is distributed[1]

[1] shared between two or more machines

# Example RPC header

```
int doSomething(int a, Object b) {
  if((this.__dj_class_mode & 0x40) != 0) {
    // this can lookup where this method call
    // should run perform the method call
    // and then return the result
    return (int)RPCHelper.doRPC(this, "doSomething",
                                new Object[]{a, b});
  }
  // body of method
}
```

- Each method converted to RPC could check a different bit in the __dj_class_mode field
- This rewrite happens gradually as the system decided to convert methods to be RPC

# Example Array rewrite

```
// before
int a[] = new int[5];
a[4] = 6;

// rewritten
dj.arrayclazz.int_1 a =
  dj.arrayclazz.int_1.newInstance_1(5);
a.set(4, 6);
```

- The implementation of `newInstance`, `set` and `get` can be anything that behaves like an array.
- For example `set` could check a bit to see if the array is in some distributed mode, and then perform a lookup of where the cell 4 is located.
- This rewrite always takes place since we can't change the type of an array instance after it is created

# Distributed object GC

- Current plan is to implement a distributed reference counting system
  - A node can use weak pointers to track when a node has lost references to an object
- If the system was to have some scavenger like GC then would likely have a lot of network communication to find objects that can be GC
- Local GC is still handled by the JVM and should continue to perform similarly.

# Currently Missing standard Java components

- Program defining class loader (common in spark like systems, even some unit test frameworks)
- Distributed IO (filesystem, network)
- Weak pointers
- Reflection (partially broken, names are getting mangled)
- GUI frameworks/crypto/"less core libraries" (may work using native bridge, not tested)
- Unsafe (used for performing direct memory access, need to handle when the object is remote).

# Distributed IO question

- Q: How to expose network sockets to a program that is running on multiple machines now
- if you open a network connection should it always redirect through the same machines (same ip)
- if open a listening socket should that just start on a random machine, or all machines
- Q: with the file system, where should new files be created
- Could augment file system api with something like `/machine-$id/that-machines-fs` for fs names

# Libraries that would like to get working

- Jetty/Tomcat: Http server, lots of things using http
- JBlas: Lots of scientific computation are at their core are wrapping Blas
  - Is really just a bunch of native method calls to C code and then uses a provided C blas library
- JUnit/some unit testing: would be nice to just run the unit tests of existing programs
- Akka: Communication framework, should replace with DJ interfaces

# Types of targeted programs

- Distributed scientific applications
- Combining and splitting service oriented applications
- Edge computation when there is some $\delta$ time delay between the edge and servers
- Easily write distributed applications using this as a base framework

# Current Scientific applications

- Commonly written in low level system languages. (Not many in Java/managed)
- Currently require that distribution managed explicitly
- Current efforts to manage distribution
  - Chapel, UPC: distribution of data is part of the language, but still require annotation
  - Grappa: Small computation that moves towards the data

# DJ with scientific applications

- Able to write the program in a higher level managed language
- Same code that was written for a single JVM can now be used on a distributed system
- Data and computation can be relocated during the running of the program
  - "More general method" that can simulate how Grappa works with moving computation towards data
  - Data placement can also be controlled like UPC/Chapel

# Current service oriented setups

- Every service is deployed as its own application
- Some RPC/serialization layer between services (protobuf, thrift, etc)
- One application per machine (virtual machine/container)
    - Communication still taking place over network interface using the RPC layer

# DJ with service oriented setups

- Would write a small inner communication management framework on top of DJ
  - All services would communicate with this simple layer rather then using the RPC library
- All services could start by running in the same JVM, would avoid communication/seralization overhead
- As a service needs more resources it can be moved to a new machine (splitting the application)
- As load drops, the program could be recombined (less resources used, less RPC overhead)
- Splitting an application need not happen at the obvious boundaries
  - Eg: if caching application, could have a bloom filter on one machine and the data on another

# Current edge computation

- Mobile application caching some data from a server side
- Web pages with CDN caching only static content
- NoSQL with distributed database at the edge
- Game server maintain all the state at a centralized location

# DJ with edge computation

- Basically two or more sets of resources that you want quick access to
  - Central database
  - Users GUI
- Edge computers may have different communication with centralized database, memory resources, processing resources
- Maybe some intermediate place which can provide high computational/memory resources that is near the edge (FOG)
- Placement of memory/caches/computation is automatically handled

# DJ as a base framework for distributed applications

- Currently to experiment with a new type of distributed application (Map reduce, graph processing, etc) have to write a lot of networking and resource management code
- Think similar to what Graal/PyPy have done dynamic languages, DJ is doing for distributed programs

# Example code for Map reduce using DJ

```
objects.par.map(obj => {
  // map opertation
  (map_key, map_value)
}).groupBy(_._1).map(objs => {
  // objs._1 == map_key
  // objs._2 == map_values
  for(value <- objs._2) {
    // do something with a value
  }
})
```

Using simple language constructs of scala's `.par` to perform the map operations in parallel over the data set of objects.
This code could easily be run on a unmodified JVM and would run the computation across multiple threads instead of multiple machines.

# Code that currently works on DJ

```
// wait until there is a second machine running
while(InternalInterface.getInternalInterface.getAllHosts.length
      == 1) {
  Thread.sleep(1000)
}

for(h <- InternalInterface.getInternalInterface.getAllHosts;
    if h != InternalInterface.getInternalInterface.getSelfId) {
  val future = DistributedRunner.runOnRemote(h,
    new Callable[Int] {
      override def call = {
        // do computation
        123
      }
    })
  println("got the value "+future.get)
}
```

# Next steps

- Larger programs/more data
- Fuzzer to find errors when running distributed
- JIT interfaces to manage the distribution of the program
- GC distributed objects