DJ: Distributed JIT

(Preprint, here be dragons)

Matthew Francis-Landau, John Kubiatowicz
UC Berkeley
{mfl,kubitron}@cs.berkeley.edu

Abstract

We present DJ our attempt at a Distributed JIT framework which allows for distributed computation and memory to be treated as fluid that can be sloshed around between available machines for optimal performance. Unlike traditional JITs, which exhibit close coupling between code generation and optimization policies, we have a unique two level approach where policies receive event notification callbacks from the runtime engine.

1. Introduction

Given the abundance of cloud platforms and massive datasets, distributed system are becoming common ubiquitous for managing these new requirements. Unfortunately, the current landscape of distributed system leaves a significant burden on the programmer to reformulate their program inside of new programming paradigms. Instead of requiring time consuming and error prone manual rewriting efforts, DJ attempts to allow the programmer to write their program as if they are targeting a single host and allow DJ to automatically determine optimal memory placement and distribution methods using JIT methods such as tracing **??**. The main features that the JIT has to work with automatically converting methods to remote procedure calls (section 4.5) and relocation of objects (section **??**) both of which are designed to facilitate reducing the distance between computation and memory.

2. Previous work

There currently exists a wide verity of distributed programming systems and platforms. The main differences between systems can be categorized into ability to recover from machine failure,[1] expectation of homogeneous cluster, requirements for specialized hardware such as infiniband (**?**), ablity to tolerate network delays.

High Performance Computing (HPC) type distributed computing typically expect a homogeneous cluster with advanced hardware such as infiniband which allows for low latency direct memory access between machines and direct memory copies without expensive serialization procedures. For dealing with failures, these systems will typically depend on checkpoint restarts which requires that the probability of failure between check points to be fairly small.

In contrast to HPC system, cloud based systems have typically are designed for commodity hardware with an expectation that a single node will fail often. This has lead to the development of data processing techniques such as map reduce 2.1 which are designed to track which computation has been lost due to node failure and reschedule it with minimal impact on the rest of the cluster.

2.1 Distributed Frameworks

Map-reduce (Dean and Ghemawat 2004; Zaharia et al. 2010) is one of the most common distributed architectures used in cloud computing for batch processing. It requires that all computation be structure in a series of Map and Reduce operations, where the Map operation takes an unit of data and maps it to a key-value pair. The reduce operation then operates on all values mapped to a single key. Multiple map and reduce operations are usually stacked to perform more complicated computations. The framework prevents arbitrary communication between independent mapping and reducing elements in the cluster, however this enforced structure allows the system to easily restart lost computation using only knowledge of what key range was affected how to reconstruct the required input data for those key's tasks.

Map-reduce many also be further extended for processing streams of data by batching events into small episodes running each set of grouped events through the map reduce pipeline.

Large scale data processing techniques have been constantly improving with new custom built query and stream execution engines which are custom build and can achieve higher performance than generic map reduce frameworks. (Melnik et al. 2010; Malewicz et al. 2010; Kulkarni et al. 2015)

In addition to clustering data based off computed keys, there also exist frameworks such as Grappa (Nelson et al. 2015) which are designed specifically for targeting graph processing applications by trying to avoid relocating ele-

[1] All distributed system could recover using check point restarts, however here we we are commenting on systems which do not require a restart

ments of a graph since that will quickly lead to fragmentation and instead relocate small executable tasks when remote memory operations are required.

2.2 Distributed Languages

In HPC environments is is more common to see custom built languages such as Chapel (Chamberlain et al. 2007; Sanz et al. 2012) or language extensions to C such as UPC (Dan Bonachea, etc) which facilitate developing distributed computations. These languages tend to use distributed Arrays or other large objects are their primitives wrapping libraries such as GASNet (Bonachea 2002) which provides abstractions over copying objects from remote machines and passing messages between machines in a cluster.

Most of these language extensions require that each distributed object contain special annotation or distribution code attached to each object which controls how locality and placement will be managed within a cluster. This has the advantage in that programmers can be aware of all distributed operations that are taking place while still using natural primitives such as array read and write operations.

2.3 Distributed JVMs

Java has a long history of people trying to develop distributed JVM (Aridor et al. 1999; Zhu et al. 2002; Miyamoto and Liblit 1997). These platforms tend to work by creating a new implementation of a Java virtual machine and may operate on page instead of object level. Some advantages of using pages instead of object directly (like DJ) is that object which are commonly used together might be located in a similar place in memory due to be allocated at a similar point in time and thus by moving a page these platforms may be acquiring resources which will be accessed shortly thereafter.

2.4 Tracing JITs

The fundamental idea of a JIT is that it will identify some block for which it can apply some optimization to while allowing the rest of the program to continue executing on a slower interpreter. As JITs mature they tend to be filled with a number of specific optimizations for countless cases. A fairly generic technique for building a first pass at a JITs is to use tracing. Tracing has been successfully used by a number of existing JITs such as Pypy for Python (Rigo and Pedroni 2007), LuaJIT (Pall), and Tracermonkey for Javascript (**?**). The way that tracing operates is that it first identifies a loop that has already executed n times and we expect that will execute at least another m times so that the time spent compiling will pay off. The tracing JIT will then create a trace which will collect any information we need such as types of variables, how branches behaved, how virtual method call sites dispatched etc. This information is then feed into a compiler which is able to use this to create a compiled version of this specific trace (specializing to the types and branch directions observed etc) which fallbacks into the interpreter in the case of assumption made by the compiler is violated.

3. Motivation

Given the number current systems that currently exist and will likely be written in the future, it would be helpful if there was a unified platform which made developing these distributed application easy. This Is why DJ supports having a user supplied JIT which is able to manage how a program should be distributed within a cluster. Additionally, there currently exists a wide range of programs that are developed for a single machine but many benefit from running on a distributed cluster. In these cases it would be beneficial if there was a way to avoid performing costly rewrites and use JIT techniques to automatically identify how the program is behaving and can be optimized for running in a distributed environment.

4. Method

4.1 Distributed Objects

A core abstraction for DJ is remote objects and the ability to relocate objects after they have been initially created. Considering that objects are core to a JVM based language it is important to make sure that non distributed objects (the vast majority) continue to operate in a high preferment manor.

All objects first receive two additional fields. The first being a mode field which allows for tasks to determine how an object is suppose to behave using a simple bit masks check. The second is a pointer to a distributed object manager which used to track additional information about an object in the case that it is operating in a distributed mode. These fields are added to all objects since we are unable to realistically change the size of existing objects without significant performance impact later. For performing code rewrites, we use the JVM's debugging interface which allows for dynamically reloading classes after the program has started. This functions as the equivalent of replacing method pointers in a vtable. There are some slight performance impacts for replacing class files after starting, the most obvious is that any compiled code inside the JVM that depends on the replaced class will become invalidated and will have to wait for a recompile.

4.2 Object Modes

Table 1 shows the potential modes that an object can be in. This is controlled by the mode field that is present on every object. The mode is local to a given cluster instance and controls weather we are proxying read and write requests (section 4.3) or converting method calls into RPC (section 4.5). There can only ever by one owner of an object at a point in time, however the owner can be moved between machines as controlled by the distributed JIT.

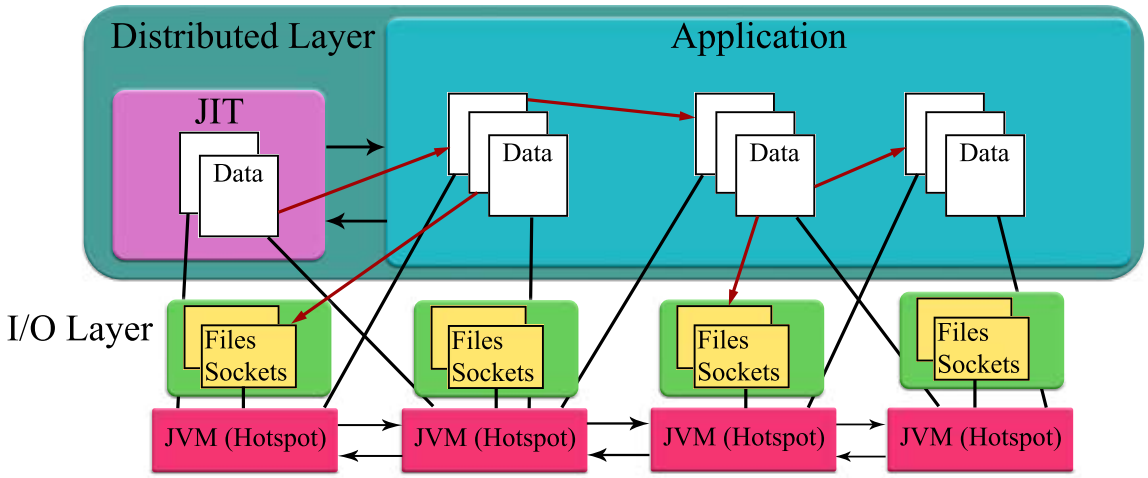

Figure 1. Overview of the DJ architecture with the majority of the application running in a "distributed layer" which appears unified between multiple Java virtual machines (JVM). The distributed JIT is running in the same distributed layer as the application and thus able to directly interact with objects of the application and receive notifications about the applications performance. (section 4.8) The I/O Layer represents wrappers for files/sockets/etc which are difficult to manage from an abstracted distributed environment. (section 4.6) Red lines represent hypothetical pointers between objects and black lines show which JVM owns a particular object (data).

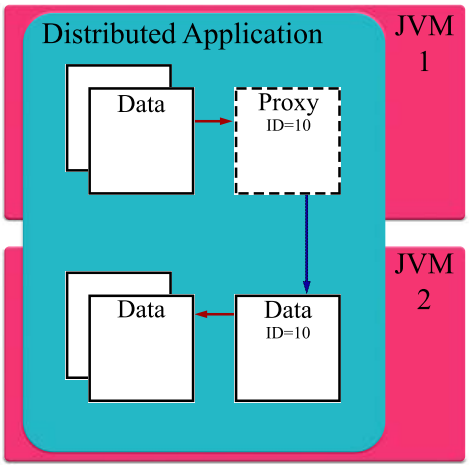

Figure 2. Remote representations of objects

Mode	Checked Operations	Intercepted		
		Reads	Writes	Calls
No instances of class distributed (Default)	-	-	-	-
Another instance of class distributed	**X**	-	-	-
Distributed (Owner)	**X**	-	-	?
(Proxy)	**X**	**X**	**X**	?
Caching (Owner)	**X**	-	**X**	?
(Proxy)	**X**	-	**X**	?
RPC (Owner)	**X**	?	?	-
(Proxy)	**X**	?	?	**X**

Table 1. Potential modes that an object can operate in. This is transparently changed by the runtime as an object converts from a local only object into distributed or RPC.

4.3 Redirected variable writes

One of the core features of DJ abstraction over a distributed system is to allow for remote reads and write operations to operate the same as local operations. We accomplish this by performing automatic rewriting read and writes of class instance variables. Ever read and write call site is automatically replaced with a static method call as show in listings 1,2. This static method is trivially inlined by the JVM since there is no ambiguity due to virtual method look ups and the corresponding static method is below initial inlining thresholds. Replacing all reads and writes with this static method during the initial conversion allows us to at a later time easily replace all read and write operations on a class without tracking and replacing all other classes which many directly read and write public members on a class.

Once there exists one instance of an object that is distributed, we have to replace a classes such that it checks whether or not it should perform remote read operations. This is due to there being a single instances of a class's code and vtable. The replaced static method is similar to listing 4 which will check the mode flag present on an object and if a specific operation requires performing additional operations and if so will call into a generic read or write handler which will communicate to the owning machine to perform the read or write.

Listing 1: Original code

```
a = 5;
```

Listing 2: Automatically rewritten to

```
ClassName.write_field_a(this, 5);
```

Listing 3: Corresponding static methods for not distributed objects

```
static void write_field_a(
        ClassName obj, int val) {
 obj.a = val;
}
```

Listing 4: Static method once distributed

```
static void write_field_a(
        ClassName obj, int val) {
 if(obj.__dj_mode & REMOTE_WRITE) {
  __dj_remote_write_I(obj, val,
    123 /* field uid */);
 } else {
  obj.a = val;
 }
}
```

Figure 3. Representation of variable rewrite transformations using static methods that are easily inlined by the JVM.

4.3.1 Proxy objects

Proxy objects are the simplest types of distributed objects available to a distributed JIT. These types of objects are automatically created via remote memory operations. They do not contain a copy of any information and only the unique identifier which serves as a distributed reference. Since these objects are empty, this means that all operations performed on them (reads and writes) will invoke remote memory operations.

4.3.2 Caches on objects

For objects that are mostly read and sparsely written, it is reasonable to activate caching on an object. This causes the runtime to maintain a complete copy of all the objects fields. This means that read operations are able to be performed directly without being intercepted by the runtime, however all write operations, even those performed by the owning machine have to be intercepted so that to inform all cached copies of a object that there was an update.

4.3.3 Distributed locks

One of Java's primitives relating to objects is a monitor that is attached to every object which can be activated through a `sychnromize` block. These blocks when translated to bytecode consist of a `monitorenter` and a `monitorexit` instruction which we are able to replace with a inlinable static method call to our own custom implementation. Given the versatility of Java's locking constructs, we always acquire the lock from the owning machine since performing any sort of optimistic locking at our level of abstraction with respect to code generation is difficult without additional support from lower level mechanisms.

4.3.4 Overhead for non distributed objects

While DJ does utilize class reloading mechanisms to selectively enable and disable features of the distributed program, there are a handful of limitations that prevent us from producing a zero overhead abstraction.

The first and most noticeable impact is the addition of two fields to all objects representing an 8 byte overhead.[2] This is mainly due to the fact that we are unable to dynamically change the size of types at runtime. This restriction is imposed by the JVM and is reasonable considering that resizing an objects would be fairly intense operation, comparable to a full GC to locate all instances of an object and adjust pointers appropriately.

In addition to the overhead imposed by extra fields, we also must be able to efficiently replace a number of operations normally performed by the Java programs. As such, all reads and writes of fields, and monitor enter and exit have been replace with static method calls. In the case that all instances of a class are not distributed, then the static methods will simply perform the read and write operation directly on an object. Given the small size of these methods, the JVM will usually directly inline these operations during first passes of code generation. Since it is difficult in many cases to check what type of object are being operated on with a monitor enter and exit bytecode, we end up replacing all instances of these instructions with our customized implementation. In the case that an object is not distributed it will be able to directly perform the locking and unlocking operations using the unsafe[3] interface to these operations. This has the unfortunate side effect of preventing the JVMs extra optimizations surrounding monitors on objects in all cases.

[2] These fields are an 4 byte integer and a Java object pointer (assuming compressed pointers of 4 bytes)

[3] `sun.misc.Unsafe`

Listing 5: Remote Procedure call check

```
public RType method_name(
    int arg1, Type2 arg2) {
  $args = new Object[] {arg1, arg2};
  if(ClsName._rpc_conf_method_name &&
     __dj_check_rpc($args,
       ClsName._rpc_conf_method_name)
  ) {
  return (RType)__dj_rpc_method(this,
    "method_name", $args);
 }
 // . . .
}
```

Figure 4. Additional check for RPC enabled methods.

4.4 Thread scheduling

Scheduling and placement of threads on creation is one of the easiest ways that a distributed JIT is able to distribute workloads within a cluster. First it is important to understand that in Java, a threaded implementation will have to contain all references that it will use within some pointer in the thread's class. This allows for DJ to determine all possible objects that a thread will start out referencing simply by looking at all references inside the thread object. To intercept the thread creation requests there are manually rewritten implementations of `java.lang.Thread` other worker pools such as `java.util.concurrent.ForkJoinPool`. There implementations are designed to redirect requests for into the runtime engine which then queries the distributed JIT for where a thread should be placed.

4.5 Remote Procedure calls

While DJ is able to schedule threads and worker tasks during their creation, we also would like to be able to migration computation in the case that this is more efficient then migrating any relevant data. Unfortunately, given that we are operating on top an unmodified JVM, it is unpractical to migrate existing threads efficiently.[4] Instead, we provided the ability for the system to dynamically set methods to behave as an RPC method call. This is accomplished by the distributed JIT informing the runtime about which method it believes should be an RPC method and which should be used when to control the redirection.[5] The runtime will then inject a check into the top of the method similar to listing 5.

[4] Projects such as Javaflow (jav) and Quasar (qua) provide potential solutions to serializing threads in Java, however both have significant performance impacts due to the need to track all stack variables.

[5] All parameters to a given method may be spread through the cluster, so the JIT is able to choose one parameter (or `this`) for which the machine that owns the object will execute the method

4.6 I/O layer

Managing I/O transparently in a distributed system such as DJ quickly adds a significant amount of overhead. This is easy to see when considering cases such as networking sockets. Suppose we have a system which simply opens networking sockets from what ever machine in the cluster issues a request. In this case, requests from the same program would appear to be coming from a wide number of IPs and hosts within the cluster and potentially break the perception that the application is running on a single host. Additionally, the networking socket essentially becomes an unmovable resource[6], which means that as the corresponding computation and data is move throughout the cluster, we would essentially have to forward all operations through what ever machine owns this I/O resource.

To deal with this there exists special classes denoted as managing I/O with a `@DJIO` annotation. Every constructor must then contain a field to identify which machine in the cluster will own a specific I/O object. These I/O classes are then allowed to access full resources on the local machine without any concern for the rest of the distributed cluster. Data can be easily shuttled between the I/O objects and the rest of the distributed application. Additionally, having the I/O objects explicitly tied to a given machine allows the distributed JIT to be aware that some objects will be unmovable and that co-locating computation near those unmovable objects may be advantageous. This separation allows for code that is closely tied to the operation of a given I/O resource to become fixed on the same machine that owns a resource while allowing compute and data access oriented code to flow between machiens in the cluster.

4.7 Garbage collection

In dealing with distributed objects, we have to consider how to collect objects which are no longer being references by the system. Garbage collection mechanism which require searching the entire heap space for all references towards an object become impracticable in a distributed system. This is due to the number of messages that would be required to identify when an object is still referenced by a remote machine, and all the objects that are in turn referenced by that object.

To counter this issue, we maintain a reference count of the number of proxy instances of some object in a cluster. Distributed objects are then tracked using Java's reference queue mechanisms which allow for receiving notifications in the event that some object has been garbage collected on a local machine.

This technique allows for a local machine to maintain as many references to a single proxy instance without having to update the referencing count while only performing refer-

[6] Migrating network connections while open would require additional support from networking hardware and the operating system

ence counting updates in the case of creating and destroying an object.

4.8 Distributed JIT

DJ runtime engine is able to interface with any distributed JIT which implements the distributed JIT interface shown in appendix A. The JIT ends up receiving notifications everytime that there is a remote read or write operation or a RPC method is activated. Additionally, it is queried when a thread is being created and required to schedule queued work. Each operation receives a self reference which is the object that that is currently the subject of the remote read or write operation. The JIT can then perform any action on this object such as querying its class using java `getClass` method or query for ownership information using the JIT commands interface (appendix B). The JIT also recieves a StackRepresentation object with events which represents where in the program a remote operation was initiated from. StackRepresenation objects can then be further collected by a distributed JIT to identify common places where optimizations could be applied.

As show in figure 1 the distributed JIT is running in the same program space as the distributed program. That means the distributed JIT will end up receiving notifications about its own remote access operations. While the JIT can make use of distributed shared objects like the rest of the program, the DJ runtime provides special annotations for explicitly controlling communication as performing unmanaged remote communication within the JIT itself tends to cause negative performance impacts for the rest of the program.

5. JIT Methods

Given the generic abstraction for our distributed JIT, we experimented with a handful of JIT policies.

5.1 NOP JIT

Our baseline JIT is a simple NOP JIT which performs no optimizations and no data relocation. The implementation of the JIT consists five empty methods which receive notifications from the runtime and two methods with trivial implementations for scheduling new threads and tasks in the cluster. The performance of this method is fairly poor and can be understood by looking at figure 5 where all read and write operations require a network communication and when running on a typical cloud network at latencies averaging 0.3 milliseconds the perform impacts quickly adds up.

5.2 Move on first access

Moving data based off its most recent access is a fairly common method and has seen use in previous distributed JVMs (Aridor et al. 1999). The idea behind this method, is that while the first access to an object will be slow, subsequent access will have the data local and thus perform much faster

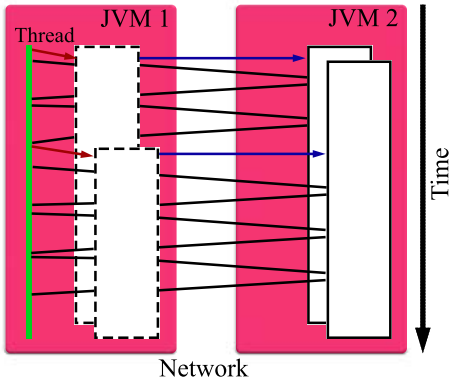

Figure 5. Example run with no object or thread movement from JIT. All objects on JVM 1 are "proxy" objects (section 4.3.1) which are dynamically created as they are first referenced. Since they are proxy objects all remote read and write operations result in network communication.

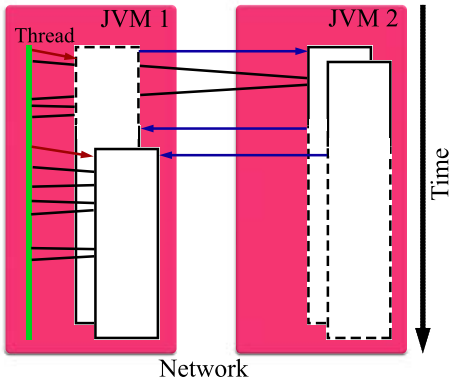

Figure 6. A more intelligent JIT moving objects when they are accessed as well as commonly accessed next objects. Since both objects are most recently accessed from JVM 1, the JIT has relocated these objects to prevent further network communication.

accesses. This simple addition significantly improves performance which is discussed further in section **??**.

5.3 Move on first access and predict next access

In trying to perform JIT like optimizations, we will utilize our knowledge about past memory access operations and order to identify which objects are likely to be referenced next. Using this information we are able to able to concurrently request multiple objects to be migrated and thus only suffer the impact of a single network operation as show in figure 6. In some cases this JIT is able to provide performance improvements since it will reduce the amount of waiting time, however it tends to cause a higher variation in the request times since objects end up being moved more frequently between machines which can cause increased number of network op-

erations in maintaining the global state of where objects are located.[7]

6. Experiments

Our experiments were run on the Google cloud computing platform using a single availability zone. The network performance is comparable to that of a typical cloud platform. A simple ping test with no other network traffic has an average ping of 0.356 milliseconds and a maximal ping of 0.701 milliseconds. An rsync transfer of 3 GB files between two instances had an average transfer rate of 129.92 MB/s.

Our programs were designed to simulate common cloud computing operations. We simulate a distributed graph computation operation by constructing a large tree object and performing depth first searches. We also simulate a simple web application using Jetty via the I/O Layer to receive network requests and then communicate requests into the distributed layer for handling the corresponding businesses logic.

7. Results

It is important to understand that the DJ platform unlike typical JITs which focus on optimizing execution speed and instructions executed by the processor, it instead focuses on memory access performed by the processor. For a program running on a typical server[8] a memory may take between 10 and 100 cycles depending on the availability of some memory in the cache. This same operation when performed remotely takes about 0.3 milliseconds due to the round trip network communication time. This means that for a thread to successfully execute at 50% speed of a single host, it must successfully predict and perform locally about 90000[9] memory operations. Successfully predicting even a fraction 900000 memory operations ends up being severely difficult except in the most trivial embarrassing parallel[10]. In practice predicting this many memory operations successfully may be impossible for a general program which is likely comprised of unpredictable hash table look ups, branches and method dispatches.

7.1 Typical Concurrent code

DJ targets programs which have been written for a concurrent environment but on a single host. These types of programs may have code similar to listing 6 where Java's synchronize primitive is used to acquire a lock on some shared map structure. This code may represent about a dozen read and write operations in addition to the acquiring the lock,

Listing 6: Realistic concurrent map lookup

```
Object res;
synchronize(map) {
  res = map.get(id);
}
```

Figure 7. This is a simple concurrent map lookup operation that one would expect to find in a typical Java program. A potential case for code like this could be session lookup within a webserver.

this means that the time spent in the critical section may be on the order of hundreds of cycles. In directly converting this block of code onto a distributed system, the lock operation now becomes a distributed lock which will take on the order of milliseconds to acquire in the non contended case. Additionally, if there is a remote memory operation inside of the critical section due to a failure by the JIT or the JIT not yet having enough samples to use to optimize, this can cause the the time spent in the critical section to take multiple milliseconds which can cause the entire distributed system to stall. Given the nature of JITs these types of failures are almost guaranteed during early phases of the program.[11]

In this simplified case we might be able to use RPC methods to avoid the overhead of a distributed lock and data structure, however, that would limit the map to being present only on a single machine. While solutions may exist in this case there are many cases where one might be acquiring two or more different locks owned by different machines within the cluster, which means that the amount of time spent within critical sections will significantly increase over the single host case.

7.2 Distributed JIT development

While it is convent to think of the distributed JIT as a single unit which is operating inside of the distributed program space, this turns out to be a difficult view to take. The main problem with this view is that any time that JIT code executing on one machine wants to communication or access the data of another it incurs the cost of a remote communication. Given that the JITs remote communication time is the same as the program's, this means that we must maintain enough information locally to perform any memory movement optimizations for an executing thread.[12]

[7] Expand on this

[8] Meaning no NUMA domains or infiniband accessible memory

[9] Approximation based off the number of memory operation that a 3GHz processor could perform in 0.3 milliseconds.

[10] Meaning that the threads of the program do not perform any communication or use shared data structures, paralleling these programs can be consider trivial

[11] Remember, JITs primarily work by recording how programs have behaved in past operations and using this information to perform informed optimizations in the future.

[12] The runtime engine provides a number of specialized annotations to allow localization of variables and fine control over RPC and asynchronous operations which are assist with building JITs.

7.3 Comments on Java

clean this section up

Building DJ on top of hotspot has a handful of advantages such as being able to avoid dealing with low level code generations, however there are a number of limitations. One such limitation is how one manages memory and raw objects. If we were operating at C like level, then we would be likely serialize objects using operations such as memory copy to a networking buffer with some patching for pointers. In the case of Java, in is impracticable to directly access the layout of objects, instead we generate specialized serialization code which reads an object one field at a time writes it into a ByteBuffer. For looking up objects, we are unable to depend on the location of an object in memory since each JVM maintains its own garbage collection and thus will migrate objects as it pleases. This means that we must maintain global object ID which require a lookup operation for each access.

There are also a number of cases where objects will end up bottoming out at some Unsafe operations such as updating a field atomically or other specialized code within the Java runtime. In these cases we essentially have to manually rewrite how these objects should behave inside a distributed system which amounts redeveloping a large amount of the Java runtime.

8. Conclusion

clean

Given the gap in performance between memory accesses on a single host when compared to a multihost environment, it is unclear if the level of rewriting mechanisms present within DJ is sufficient to accomplish an acceptable program transform. Additionally, given how concurrent Java code is written as discussed in section 7.1, it is unlikely that the level of rewrites required to optimize general cases falls within the realm of current JIT technology. Still more work is required to comment on if DJ works as a sufficient platform for prototyping and building new types of distributed systems.

References

Apache Commons - Javaflow. URL `http://commons.apache.org/sandbox/commons-javaflow/`.

Parallel Universe - Quasar. URL `http://docs.paralleluniverse.co/quasar/`.

Y. Aridor, M. Factor, and A. Teperman. cjvm: a single system image of a jvm on a cluster. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 4–11, 1999.

D. Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007. ISSN 1094-3420. doi: 10.1177/1094342007078442. URL `http://dx.doi.org/10.1177/1094342007078442`.

Dan Bonachea, etc. Berkeley Unified Parallel C (UPC) Project.

J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9.

G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2.

S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.

C. Miyamoto and B. Liblit. Themis: Enforcing Titanium Consistency on the NOW. `http://www.cs.berkeley.edu/~liblit/themis/`, Dec. 1997. CS262 semester project report.

J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL `https://www.usenix.org/conference/atc15/technical-session/presentation/nelson`.

M. Pall. The LuaJIT Project. URL `http://luajit.org/`.

A. Rigo and S. Pedroni. JIT compiler architecture. Technical Report D08.2, PyPy, May 2007. URL `http://codespeak.net/pypy/dist/pypy/doc/index-report.html`.

A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, A. Navarro, V. Litvinov, S.-E. Choi, and B. Chamberlain. Global data re-allocation via communication aggregation in chapel. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 235–242, Oct 2012. doi: 10.1109/SBAC-PAD.2012.18.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1863103.1863113`.

W. Zhu, C.-L. Wang, and F. Lau. Jessica2: a distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 381–388, 2002.

A. Distributed JIT interface

Listing 7: Distributed JIT interface

```
public interface JITInterface {

    // Constructor is call on the main node before the main method is called

    // a new client has been created
    // this is called on the main node
    // the id can be used with the distributed running to run some
    // code on a targeted machine
    void newClient(int id);

    // These are called on the machine that invoked the remote operation
    // the self object can be inspected to see what state this object is now in
    // eg where it is located, or attempt to find out what else it references
    void recordRemoteRead(Object self, int from_machine, int to_machine,
      int field_id, StackRepresentation stack);

    void recordRemoteWrite(Object self, int from_machine, int to_machine,
      int field_id, StackRepresentation stack);

    void recordRemoteRPC(Object self, int from_machine, int to_machine,
      StackRepresentation stack);


    // these are recorded on the machine that currently owns the object self,
    // this will make some policies easier to implement such as always move
    // an object to the machine that performed the last read
    void recordReceiveRemoteRead(Object self, int from_machine,
      int to_machine, int field_id);

    void recordReceiveRemoteWrite(Object self, int from_machine,
      int to_machine, int field_id);

    // called during Thread.start from the machine that is starting the thread
    // self is the runnable object that was passed to the thread to start
    // a trivial implementation would just return from_machine
    int placeThread(Object self, int from_machine, StackRepresentation stack);

    // when something submits work into a work queue like system
    // eg using .par on a scala collection
    void scheduleQueuedWork(Object self, int from_machine,
      StackRepresentation stack);
}
```

B. Distributed JIT operations

Listing 8: Distributed JIT Commands

```
public class JITCommands {
    static public int getObjectLocation(Object self);

    // move the ownership of some object to a target machine
    static public void moveObject(Object self, int target);

    // use a field reference to determine where to move an object
    static public void moveObjectFieldRef(Object from, int field, int target);

    // make a read only cache of some object on target machine
    static public void cacheObject(Object self, int target);

    static public void removeCacheObject(Object self, int target);

    // send a command to the runtime to rewrite a method to have the RPC header
    // and reload the class with the new code
    static public void makeMethodRPC(String clsname, String methodSignature,
      int argumentPosition);

    // identify what the current configuration is on a given method that has
    // rpc enabled, eg the argumentPos value
    static public int lookupMethodRPC(String clsname, String methodSignature);

    // when the system get some work submitted to something like a ForkJoinPool
    // it is the job of jit to schedule when the jobs should actually be run
    static public void runQueuedWork(Object self, int target);

    static public Object lookupObject(byte[] identifier);
}
```