Intro[1]

RudolFLabs seeks to evangelize functional programming and dynamic languages to the wider scientific community - and recently has recognised the value of deep understanding of the underlying implementation and architecture of the JVM interpreter.
The problems with memory allocation and the garbage collector algorithms are well known to any experienced programmer and we seek to clarify with design patterns how to resolve memory leaks and optimise memory use. There are design issue's[2] in the implementation of 4th gen languages using the JVM as an interpreter. In particular the issues that arise specifically with the heuristics of the JVM's runtime Just In Time (JIT) compiler. The recent addition of the "invokedynamic[3]" opcode shows the stimulus to evolve the instruction set available to meet the demands that 4th gen languages place on the JVM

This work seeks to encapsulate in a format accessible to scientists writing functional code designed to run on a JVM a set of design patterns/anti patterns to produce code that is optimised for the JIT heuristics.

Background

In 1996, C programmers built a virtual processor sometimes referred to as a Java Virtual machine, but in reality was a simple LIFO data structure running on top of a little known C runtime called Greenfields, the documentation of which has never been located, demonstrating a somewhat ad hoc nature of the early team synchronisation environments in Java language development.

The efforts of James Gosling to evangelize a 3rd generation internet language led to what is now termed as Java. Unfortunately coincident with the commercialisation of the internet everything was thrown together too quickly and many design mistakes/compromises[4] were made which has led to many many problems in the Java Language specification and JVM architecture. Other, more robust mature functional languages such as LISP are now gaining a wider acceptance in commercial development houses and variants are being implemented to run on the JVM. This has been enthusiastically received especially by programmers who now have a chance to design applications in the clean, terse compositional functional style rather than the thousands of lines of repetitive Java boilerplate.

Java Class terminology

---

[1] This doc came together as a journal diary to prepare a thorough syllabus. It took a tangent when I stumbled across the 4th gen hot code issue. I have finished this now as I have everything that is important for a beginning understanding of the JVM

[2] a JVM limitation is that the size of a method can not exceed 65535 bytes

[3] http://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html

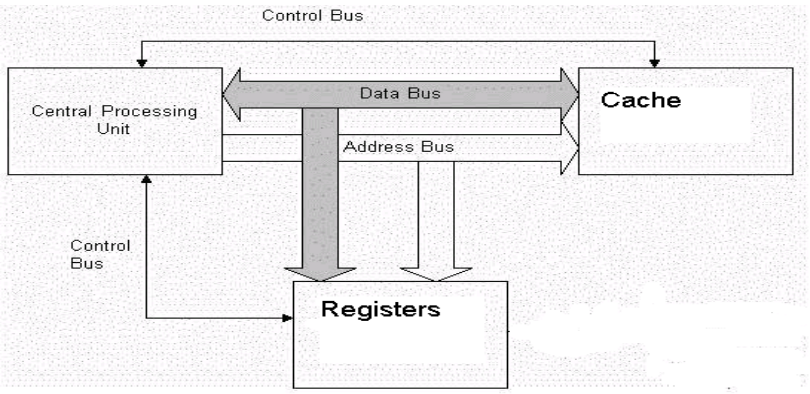
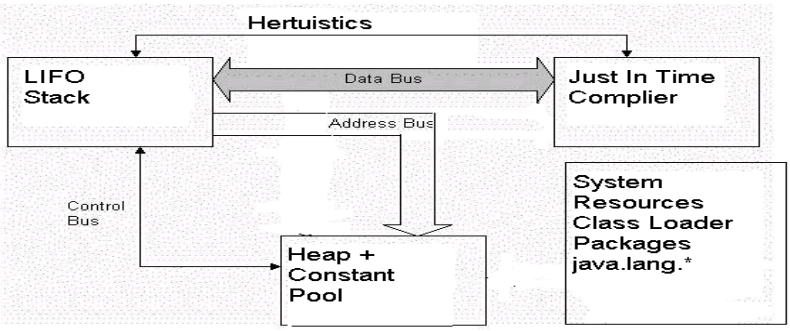
[4] exceptions for example

Hot code is code that is optimised for the Java Just In Time compiler (JIT).

Hot complex code is code that sets-up JIT for method inlining in a context where without the complexity the JIT would serve up the code to the Java runtime stack without the method inlining.

JVM Analogy

Below top is a schematic for a primitive cpu first used in older computers from the 1970's, it is helpful to see the components and how they work together, for instance the data bus that connects the registers to the processing logic unit. In modern designs the registers are fabricated into the cpu substrate as integrated logic gates for performance and increase in data bus speed. It is helpful to see the old design as it emphasises the issue connected to the addressing of memory locations.

Simple CPU Cache is fast memory, Cache serves up data faster and boosts overall performance	
JVM Analogy here the JIT is acting like a cache, serving up Java bytecodes and boosting overall performance the heap is the slower register memory	

JVM is a LIFO stack-based machine

- Each thread has a JVM stack which stores frames
- A **frame** is created each time a method is invoked, and consists of
  - an **operand stack**,
  - an **array of local variables**,
  - and a **reference** to the runtime **constant pool** of the class of the current **method**.

Summary

Objects reside in the heap, The PermGen is a special area of the heao reserved for constsants and class variables.

- Think of constant pool as fast memory “private static final”
- **The operand stack** is controlled by **opcodes**

A software engineering perspective
The stack frame model is for a method, **frame ⇔ method**

The JVM is a C program, C is a method based language, it has object like structures, enums, while one can write objects in C, C is not Object Oriented. The JVM is optimised for methods perhaps because C programmers think in terms of methods and they wrote the JVM

When the JVM invokes a method it checks for method property flags EG “ACC_SYNCHRONIZED” a synchronised method uses this flag to gain a performance increase over a synchronise code block, in the oracle tech docs and elsewhere they often speak of virtual methods. This comes from one of the opcodes for a method call “invokevirtual” which is the opcode command to call an instance method.

1. invokeinterface method declared in a interface
2. invokespecial constructor, private, or supeclass
3. invokestatic static
4. invokevirtual instance

in the JVM spec a class has a method_inf struct (recall struct from c) the struc-info data type, this type has some lookup tables for local variables, exceptions and line numbers(unwinding, inlining) the address space is constrained by these tables to 65535

Application
Stack based ⇔ no virtual hardware cache

JIT (Just in Time) Compiler

can think of this as like a cache, recall cache is fast memory inline with cpu before registers, it serves up the bytes to the cpu faster than the register memory, however this can be problematic (think pointers), JT serves up the bytecode to the JVM in analogous fashion
JIT (Just in Time) Compiler Optimisations

Among the many JIT (Just in Time) Compiler Optimisations **method inlining** (method is broken down and recombined for performance boost) is the most important and relevant to functional programming.

JIT's routinely inline large functions (thousands of bytecodes), inline nested functions many layers deep, and even inline functions which are only known dynamically – based on type profiling information. JIT's use Class Hierarchy Analysis or aggressive type analysis to decide the possible targets of a function call, they inline multiple targets and use a runtime test to decide the correct flavor, and/or have multiple fall-back positions if the actual runtime types don't match the profiled types. Non-final functions are routinely inlined, and the JVM rolls back if the function is later overridden. An emerging issue for new functional languages running on the JVM (e.g. JRuby) and new programming paradigms (e.g. Fork Join) have exposed a weakness in the current crop of inlining heuristics. Inlining is not happening in a crucial point in hot code exposed by these languages, and the lack of inlining[5] is hurting performance in a major way. Iit's a Major Problem facing JVMs with long and far reaching implications

Megamorphic inline caching

In computing, **memoization** is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously processed inputs. There is an analogy in hardware where the cache memory boosts performance by storing data in fast memory close to the cpu, avoiding longer flow of address to bus to cpu, and the inherent time delay from this flow.

Megamorphic inline caching is a technique to drive the JIT in an analogous fashion to the hardware cache. The idea is to write code that does not break the JIT optimisations at runtime. The most likely place for code to break the JIT heuristics is an a loop. A loop can be hot, complex and megamorphic this loop will be 10x faster. Below in the table are some examples to express this concept,

High level JVM languages often produce non "hot, complex and megamorphic" slow code

[5] work around for these languages, if method is static, final inlining is guaranteed

long getALong (long a, long b) { return a|b }	ok can inline ie “HOT”

void okLoop long[] dst, long[] src1, long[] src2) {
for(int i=0; i<dst.length; i++)
dst[i] = getALong(src1[i],src2[i]);
}
	ok can inline “HOT”
	closed under addition
	
	loops are the performance bottlenecks

void okLoop (Rectangle a, Image dst, Image s1, Image s2) {
for(int i=a.low; i<a.high; i++)
for(int j=a.left; j<a.right; j++) {
idx = i*a.width+j;
if(dst.pixels) {
dst.red [idx] = getALong(s1.red [idx],s2.red [idx]);
dst.blue [idx] = getALong(s1.blue [idx],s2.blue [idx]);
} else if(dst.pixels) {
....other stuff.... } } } }
	ok can inline “HOT”
	
	despite increase in complexity everything is clear to the JIT

void notOkLoop (ObjWithFuncs fxy , Rectangle a Image dst, Image s1, Image s2) {
...
dst[idx] =fxy.call(s1[idx],s2[idx]);
...
}
	can not inline
	as result of architecture JVM should be clear now the ObjWithFuncs calls its functions in a separate frame
	
	temp just dropped a few degrees

if you use objects with functions in loops if you can make the static final the JIT often can cope and you will maybe not lose method inlining. The JIT likes everything to be static final clear deterministic primitive,

List<String> bottleNeck = …....(some list constructor)

for (int i=0; i <bottleNeck .size; i++)

is much faster than

for(String s: bottleNeck)[6],

[6] no doubt you knew this but who is going to write the first idiom?

for (int i=0; i <bottleNeck .size; i++) is loop invariant[7],
then the situation become worse for the JIT when we add objects such as Strings

for(String s: bottleNeck) while looping

if synchronised then we are increasing the chance of breaking the JIT optimisations

There is a need to take the jargon inherent in the JVM specification with a grain of salt.

Jargon	Clear and Simple
Hot methods are more likely to be inlined	JIT is lazy, needs you to do everything possible to make its job easier
Segregate fast paths from slow paths, keeping hot methods small (35 bytecodes or less) with error processing and slow paths in separate methods.	Frame table is optimised for 35 bytes JIT reluctant to inline exceptions as if ever there was an overloaded polymorphic object its Exception, and methods are going to be cut more slack by JIT than local frame objects

The chances of breaking the JIT optimisation can be further increased by the constructs in the k\language we take for granted, for by now it should be clear that throwing an exception in a method signature has to be more “Hot” than try catch.

Summary

- the JIT compiler has a bunch of tests/conditionals (heuristics) to decide if it will inline a method, your method design will fail if it can not pass these tests

- principally what is the complexity of the inlined result for a given method, how much work will it be for the JIT, the more work you do to “inline the code” the more likely you are to pass these tests
- complexity is the work you do to inline the loop for the JIT
- a hot complex loop will pass the test, no functional decomposition in loops[8]!!!

[7] hI believe that (int i=0; i <bottleNeck .size; i++) is only invariant compared to for(String s: bottleNeck), in that the engineers who write the technical docs for this stuff jare just using the “invarient” term as a code for JIT does not want to look inside the List<String> bottleNeck to gets its size, it cant be bothered to do the extra work

[8] programming using classes that contain one method that call other classes also with one method, clearly Scala and 4th Gen functor languages will break JIT aggressive inlining

- one aspect of a well-designed Java program is highly factored, fine-grained design -- in other words, lots of compact, cohesive methods and compact, cohesive objects.
- method invocations reduce the effectiveness of compiler optimization.

Below are a set of example classes that expand and demopnstrate the ideas presented in this section.

// JIT friendly functonal object

```
class HotMemoryFunctional {// demonstrate JIT friendly code
// if possible make everything here static final
// consider static code blocks for exceptions as Polymorphisim can throw off the JIT
// optimisation in a method body, staic final guarentees JIT optimisation
final FileOutputStream fHot;
static { // need this for FileNotFoundException exceptions in heap no problem for JIT
 try {
       fHot = new FileOutputStream ("testfile1.txt");
            } catch (FileNotFoundException e) {e.printStackTrace();
       }
       }// end  example only static code block

// no polymorphisim exception, will be issues with synchronised method and thrown
// exception, specifically what happens to any locks on the method when the exception
is
// thrown which comes first for lock release, frame or exception
public static final synchronized String boilingFunction(final String hotString)
throws Exception {
       // locals, try and stay inside 35 bytes of local variable memory
       String lastResort = null;   // if possible have your Strings as global final
static heap dwellers
       lastResort  = hotString;  // JIT is happy because hotString is final
       // !!!! Warning thin ice, working with Objects can break JIT optimisation
       // walk with care do everything possible to make it clear to JIT what is
happening
       final byte[] economicalDataObject = lastResort .getBytes("UTF-16LE"); //
recall 16 bit unicode
       final short twobytes = (short) economicalDataObject .length; // JIT is happy
if you do the work to help inline the loop!! want hot megamorphic loops only
             final byte[] complemnt = new byte[twobytes]; // recall 16 bit unicode
```

```
            for (int i=0; i<twobytes; i++) { // inline the loop as much as possible
for lazy JIT
            complemnt[i] = (byte) ~economicalDataObject [i];
         }
       return new String(complemnt);
       }

       public static  void main(String[] s ) {
              HotMemoryMap map = new HotMemoryMap();
              try {
              System.out.println( map.boiling("hot"));
              } catch (Exception e) {e.printStackTrace();
              }
       }
  }

 output  –ÿ?ÿ‹ÿ
```
//it is not that you have to write code like this but rather to demonstrate what the JIT compiler is

Further JIT Optimisation theory

To break down the concepts the following schema can be useful.

- programs can be "cache friendly" (i.e. there is a limited set of methods that is invoked frequently).
- monomorphic caching assumes objects, method signature, class data structure, in code will not change type , ie no polymorphism, polymorphic code will break a monomorphic caching scheme
- polymorphic caching: converse, increase in overhead, system finite => upper bound
- when polymorphic caching @ upper bound we say scheme has become "megamorphic" and no more inline caching is performed.

one possible gross simplification of this compiler schema is that a "megamorphic" cache is an optimised “polymorphic” cache, that is a polymorphic compiler optimization that in its steady state is at its upper bound for performance.

Where a polymorphic cache scheme will not break with polymorphic code.

Summary

write optimised for cache code

- 1, there is a limited set of methods that is invoked frequently). (inline optimised)
- 2. ditto polymorphism, this is just common sense when you think about it, of course there will be an overhead when the JIT compiler has to figure out the type of the object from its superclass?
- ` as regards 2 there is an extra dimension as not only are we thrashing the compiler with polymorphic inheritance schemes we are potentially losing the cache with a consequent big runtime hit.

Caching scheme have always been problematic. Experienced engineers will speak of well balanced systems. There is an optimised relationship in cached systems. While no cache is slower there is an an upper limit to the cache memory, if you exceed this limit you can actually harm performance.

Overall we need to get a feel for writing well balanced code.

Java Class Loader

- ClassLoaders are meant to abstract the process of obtaining classes and other resources. A resource does not have to be a file in a filesystem; it can be a remote URL, or anything at all that you or somebody else might implement by extending `java.lang.ClassLoader`.

- ClassLoaders exist in a child/parent delegation chain. The normal behavior for a ClassLoader is to first attempt to obtain the resource from the parent, and only then search its own resources—but some classloaders do the opposite order (e.g., in servlet containers).

- ClassLoaders are used to, well, load classes. If your program can write files to a location where classes are loaded from, this can easily become a security risk, because it might be possible for a user to inject code into your running application.

Java Operating System JOS

In operating systems, the **process** is the abstraction that separates the kernel from applications and applications from each other

in the operating system view the JVM is the **process** as it performs type-safety, language-level access control {verifiers for compiled bytecodes) and modifies the larger system(eg objects on the heap)

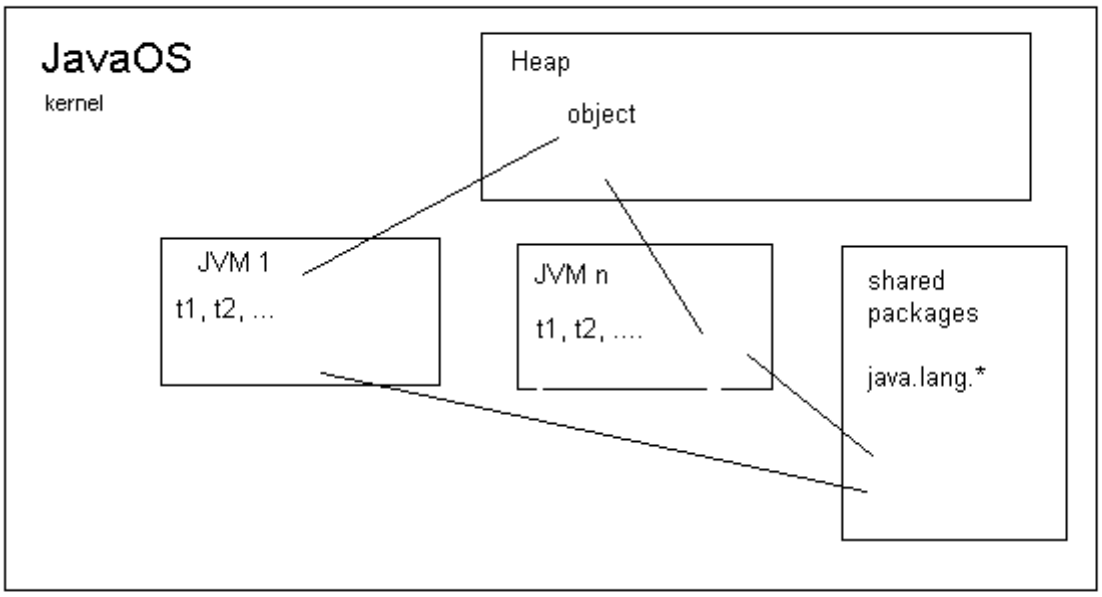
Any op-sys with a single process is trivial, so research for the JavaOS is to provide a framework for multiple JVMs running inside a shared resource collection(concurrency on steroids)

the system resources, heap, package libraries (like java.lang.*) are to be shared to avoid duplication of resources

JVM was originally written using a multi-threading C runtime called *Greenthreads*

runtime coordinates activities, such as I/O, between interpreted Java threads and the host system.

The Java OS has the following memory model JMM.



tk is kth thread

Class Memory show a layout of the memory map for the JMM.

```
class MemoryMap {
static final …..//  everything here is in constant pool

Object cold; // not static final private on the heap
void cold1() {  // everything here in frame1 = f1}
void cold2() { // everything here in frame2 = f2}
}
```

There are many Java-based projects that provide OS-style support for Java applications in a single virtual machine:

Some notes on the standouts, the research all seeks to deal with the problems of running multiple applications inside a single JVM MemoryAccessErrors MAE

the J-Kernel, this solution seeks to address concurrent access, time evolution of object life cycles in the heap, garbage collection and the fact objects with references to other objects when on the stack may point to objects that are garbage collected leading to MAE, often systems will not only fail to recover from MAE's can be worse for magnetic disks, blue screen of death

KaffeOS, uses run-time write-barriers to prevent interheap references between processes. separates each Java process's heap and supports independent garbage collection of each heap. is best opensource implementation in my humble

the Princeton JVM, a multiprocessing JVM developed to explore the security architecture ramifications of protecting applications from each other (concurrency on steroids?), as opposed to standard j-sec which protects system resources

Sun's JavaOS, is a poor closed source project, but the first, now Sun has JavaOS for Business, only runs a single Java application; JavaOS for Consumers, for embedded systems., JavaOS for Network Computers

Both of these systems require a separate virtual machine for each Java application, which results in unnecessary overhead for duplication of virtual machine structures and initialization.

These models all rely on any object having a kernel state property that can be local or global, so in a gross simplification a calling JVM can give an object a local scope, or global scope etc... for obvious reasons
this kernel state property has a representation in the bytecode.
The JOS kernel itself is the usual op-system definition that allows the utilization of the underlying hardware capabilities, very similar to the JRE on non java op-sys

Do not just think of the Java Virtual Machine as a virtual device used to interpret Java bytecodes, It executes the bytecodes, handle exceptions, manages almost all of the RAM in the computer, and handle the simultaneous execution of multiple threads. In short it is an infrastructure provider, an operating system of itself.

Outside of JOS, ie typical jre on windows JVM is interpreter in the JOS the JVM does not interpret bytecode, it executes bytecodes for all this .

Object Life Cycle

The JVM is a Finite State Machine, however because of runtime compilation the JVM posses multidimensional states and hence complexity. For example there is a method/stack/frame state coincident with a object/heap memory state both put locks on system resources as outlined in the Java Memory Model JMM.
The garbage collector tries to figure everything out and maintain enough physical memory by ts reclaiming unused objects resources so just as we see with the JIT and its optimisation heuristics, the GC has algorithms it uses to decide when an object is safe to be reclaimed so just as in the JIT case it will be possible to write code patterns that are optimised for memory efficiency. As java programmers we are very good at creating objects and there are many battle tested design patterns we can use to create objects, Singletons and Factory for example, however we are lousy when it comes to destroying objects, where are the patterns for this?, where is the dumpster pattern, where is the recycle pattern?

Steps that will be performed by the JVM when an Object is created.

1. Space is allocated for the object.
2. Object construction begins.
3. The superclass constructor is called.
4. Instance initializers and instance variable initializers are run.
5. The rest of constructor body is executed

Objects that are held by at least one strong reference are considered to be in use. In JDK 1.1.x, all references are strong references. Java 2 introduces three other kinds of references: weak, soft and phantom

Anti Pattern Memory Leaks

Pattern Beware static lists

```
public class MemoryLeak {

static List<Leak> hosePipe = new ArrayList<Leak>();

static void makeLeak() {
Leak leak= new Leak();
hosePipe.add(leak);
}

public static void main(String[] arg) {
makeLeak();     // do more stuff    }
       }
```

When the makeLeak method returns, the stack frame for that method and any temporary variables it declares are removed. This leaves the Leak object with just a single reference from the hosePipe static variable, now the Leak Object will not be garbage collected as long as there is a reference to the memory leak class somewhere.

A Time series Memory map

permGen space	exempt from generational GC,	methods and types
eden space	new objects not expected to last so long	idea is that most objects only have short lives, so if new probably will need to be collected soon, high priority flag for gc
survivor space	have survived at least one round of GC	have survived so will probably last a while, medium priority
tenured generation	been around for a while	probably here for good, low priority

Some further links for the Java Language Specification JLS Object life-cycle

1. Initialisation http://www.artima.com/insidejvm/ed2/lifetype4.html

This is the important stuff http://www.artima.com/insidejvm/ed2/gc9.html

the generational model is the one everyone is using

so basic idea is
1. initialisation,
2. a generational cycle where the object can end up in a category that is in memory terms
   a) unbounded (mature object)
   b) bounded (younger generation)
3. finalisation

Cycle is init() → survival → finalize()

? Finalization of Object http://www.artima.com/insidejvm/ed2/lifetype5.html from Link1

When an object becomes unreferenced by the application, the virtual machine may reclaim (garbage collect) that memory. Implementations can decide when to garbage collect unreferenced objects--even whether to garbage collect them at all. Java virtual machine implementations are not required to free memory occupied by unreferenced objects.

closures as it stands in java now are inline implementations of interfaces, there can be problems with these constructs related to memory management and concurrency locks, more to come

Appendix Q&A[9]

q: not sure where 'hot code' fits in ... is it 1. javac, 'source code' -> 'byte code', 2. JIT compiler, 'byte code' -> 'hot code'? and is 'hot code' machine code?

JIT is part of the runtime for a JVM, hence it always runs. however in the course of its functions it can optimise the code, you can think of the runtime as an operating system kernel for the process that is the JVM. see JavaOS section for more details on this.

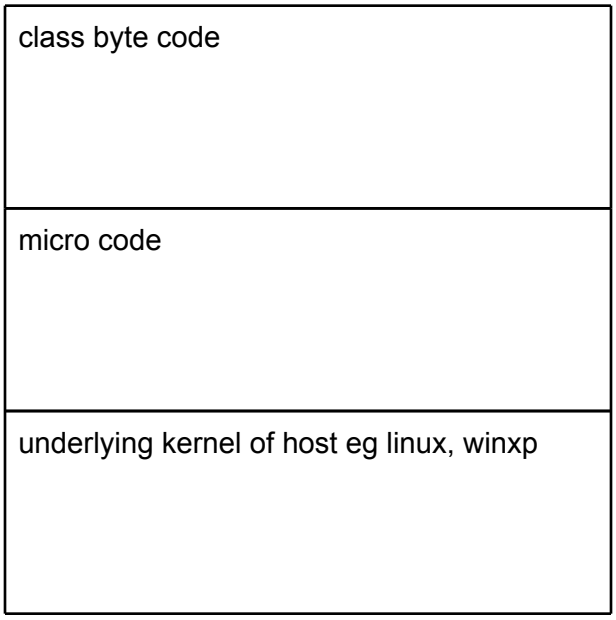
javac, 'source code' -> 'byte code', **yes**
JIT compiler, 'byte code' -> 'hot code'? **yes spot on the money**
and is 'hot code' machine code? **yes and no, it is optimised byte code which will eventually be machine code as the JIT hooks into the base kernel**

in its most general sense “hot code” is just vanilla java code that passes the optimisation heuristics for optimisation of the existing code by the JIT to 10x faster “hot code”

also I will try and expand the JavaOS section, the reference to machine code is a good one, to the best of my knowledge, I have found it difficult to find a good definitive reference online, the last time I looked at this stuff the microcode layer was there, so any machine code will be in the host opp sys kernel

Vanilla java code

[9] ask questions to stimulate critical thinking and to illuminate ideas



q elaborate method inlining.

recall everytime we create a method we create a stack and its local variable scope ⇔ frame ⇔ memory allocation
=> lots of methods => lots of stacks, thrash the JVM stack implementation process
=> lots of methods => lots of frames =>lots of memory address to track
=> lots of memory is stolen from the heap where all the big objects live
if we have a process that can reduce the number of methods we
save memory, increase processing speed as no need big look up table of memory address for local variables, free up memory for the big objects the threads are busy creating on the heap
this is method inlining, it has an associated process called unwinding java uses unwinding for exceptions way to much

q: does the JIT compiler do a byte code to byte code transformation? what is its input and output?

These question illustrate why the JavaOS section is included.xed size stack

1.The kernel of the base opp-system ultimately translate everything down to machine code
2. the jre is the bridge if you like between the underlying base kernel, but at the same time it is a kernel itself for the JVM
3 the JIT is a component of the jre so the byte code instructions will eventually be translated as the base kernel machine code
3. In many implementations the JIT will break out of java bytes to machine bytes optimised for for the underlying hardware this is not guaranteed
what is its input vanilla java byte code
output **can be anything** depends on the context

eg method inlining: will output java bytecodes for the method,
basically the JRE will spit out machine code for the JVM process and the JIT facilities this but is embedded as part of an opaque kernel process

think
optimisation <=> java bytes
runtime kernal process <=> machine bytes

q:what's the limit to how much can be inlined? how aggressive is the inliner? does it often inline stuff even when that makes the performance worse?

I have not found any mention of a limit in the oracle docs, however they routinely speak of processes where the inline 10's of thousands of bytes, if you take the optimal frame memory space of 35 bytes that's a lot of methods

does it often inline stuff even when that makes the performance worse?
this is the point of the heuristics, is very aggressive, if method code passes the heuristics inlining will proceed

q: what does 'megamorphic' mean?

is technical term from compiler builders lexicon. is very difficult concept to get across for me not being a compiler guru but I have an analogy that works for me and helps me get a handle on a lot of the other stuff as well

analogy is with hardware cache, where when you add fast memory inline with the cpu to store bytes of data and memory addresses you get a big performance boost, but there is an upper limit imposed by the cost to the cpu for running the cache

monomorphic assumes simple data, polymorphic assumes complex data but requires more resources, just as in the physical there is an upper limit imposed by the burden of running a polymorphic scheme.

When the finite state machine(JVM) is running at its optimal performance with a polymorphic scheme we say the cache is "megamorphic"

at the moment this is all I have, please don't quote this until I have a chance to do some more reading. This is condensed from what i included in the doc about this above

if you have loops with objects calling methods in those loops, and those methods are overloaded and the objects themselves are polymorphic then a loop like the is “not hot complex megamorphic”

Exception, and methods are going to be cut more slack by JIT than local frame objects
q: Can you clarify? This is interesting
this answer also addresses
would be interesting to see how expensive throwing, catching, and propagating are

is difficult to justify this statement at the moment, it just comes from reading the oracle tech docs

1. when an exception is thrown the JIT “unwinds” the bytecode that led to the exception being thrown so when u get the stack trace thats says “caused by >>>> called at line 132 in class Dodgy.java ,,,,,,, “
this is a result of the JIT unwinding or rolling back through all the code to the source of the exception …

as a result of inlining the bytecode sequence can have 10’s of thousands of bytes and somewhere in there is a divide by zero or a null pointer call, the JIT has to wind back through all the bytes to find the source

if the method throws the exception and I am not 100% on this but just as is definitely the case when you synchronise a method I think a compiler flag is set for the exception type for the method, when you throw an exception you can think of this as like setting an annotation in the bytecode, that the JIT can use to wind back immediately to that methods source in the inline sequence of bytes

byte sequence
meth1**********meth2**********

here * = bytecode, the method is invisible to the JIT is only for us to see now so you can see how the inline byte sequence is built , the JIT would only see

****
****
****

once inlined all the methods are just a mammoth sequence of bytes

but if flag/annotation JIT would see something like

****
****
@throws exception block start
****
****
@end exception block
****

I am not 100% on this but the basic idea is correct, just the actual implementation i am still not 100% sure

Links General

1. jvm specs: http://docs.oracle.com/javase/specs/

2. more about the jvm: http://www.artima.com/insidejvm/ed2/index.html

3. gc http://java.dzone.com/articles/how-tune-java-garbage

Links Specific

1 sun.misc.Unsafe.class
http://stackoverflow.com/questions/5574241/interesting-uses-of-sun-misc-unsafe

this link highlights some uses of Unsafe.class however most use cases are implementable with alternative calls, however everyone agrees can use Unsafe.class to speed up Array.copy()

2. sun.misc.Unsafe.class
http://highlyscalable.wordpress.com/2012/02/02/direct-memory-access-in-java/

this link unpacks the mechanism which allows Unsafe.class to speed up Array.copy(). recall in Java arrays are full objects, so what is true for arrays must be true for all objects?

3. Great comparison between python java re interpreter vs vm @matt
http://stackoverflow.com/questions/441824/java-virtual-machine-vs-python-interpreter-parlance