Inside a JVM -2

With Cliff Click

What Is In a JVM?

What Is In a JVM?

A whole lot!

The **V** in **Virtual** is a useful abstraction

- The JVM Provides **Illusions** (Services)
- Programmers focus on value-add elsewhere

Lets run down the set of things

- in the Java **Virtual** Machine
- and **not in** a Real Machine

Illusion: Infinite Memory

Garbage Collection – The Infinite Heap Illusion

- Just allocate memory via 'new'
- Do not track lifetime, do not 'free'
- GC figures out What's Alive and What's Dead

Vastly easier to use than malloc/free

- Fewer bugs, quicker time-to-market

Enables certain kinds of concurrent algorithms

- Just too hard to track liveness otherwise

Illusion: Infinite Memory

GC have made huge strides in the last decades

- Production-ready robust, parallel, concurrent
- Still major user pain-point
  - Too many tuning flags, GC pauses, etc
- Major Vendor point of differentiation, active development
- Throughput varies by maybe 30%
- Pause-times vary over 6 orders of magnitude
  - Old default parallel: full GC pause: 10's of Gig's w/10sec
  - Azul GPGC: 100's of Gig's w/10microsec

Illusion: Bytecodes are Fast

Class files are a lousy way to describe programs

There are better ways to describe semantics than Java bytecodes

- But we're stuck with them for now
- Main win: hides CPU details

Programmers rely on them being "fast"

It's a big Illusion: Interpretation is slow

JIT'ing brings back the "expected" cost model

Illusion: Bytecodes are Fast

JVMs eventually JIT bytecodes

- To make them fast!
- Some JITs are high quality optimizing compilers

- Amazingly complex beasties in their own rights
  - i.e. JVMs bring "gcc -O2" to the everyday Java programmers

But cannot use "gcc"-style compilers directly:

- Tracking OOPs (ptrs) for GC
- Java Memory Model (volatile reordering & fences)
- New code patterns to optimize

Illusion: Bytecodes are Fast

JIT'ing requires Profiling

- Because you don't want to JIT everything

Profiling allows focused code-gen

Profiling allows better code-gen

- Inline whats hot
- Loop unrolling, range-check elimination, etc
- Branch prediction, spill-code-gen, scheduling

JVMs bring Profiled code to the masses!

Illusion: Virtual Calls are Fast

C++ avoids virtual calls – because they are slow

Java embraces them – and makes them fast

- Well, mostly fast – JIT's do Class Hierarchy Analysis
- CHA turns most virtual calls into static calls
- JVM detects new classes loaded, adjusts CHA

- May need to re-JIT
  - When CHA fails to make the call static,

    uses an *inline cache*
  - When IC's fail, virtual calls are back to being slow

Illusion: Partial Programs are Fast

JVMs allow late class loading, name binding

- i.e. classForName

Partial programs are as fast as whole programs

- Adding new parts in (e.g. Class loading) is "cheap"
- May require: deoptimization, re-profiling, re-JIT
- Deoptimzation is a hard problem also

Illusion: Consistent Memory Models

ALL machines have different memory models

- The rules on visibility vary widely from machines
- And even within generations of the same machine
- X86 is very conservative, so is Sparc
- Power, MIPS, ARM less so
- IA64 & Azul very aggressive

Program semantics depend on the JMM

- So JVM must match the JMM
- Else *program meaning* would depend on hardware!

Illusion: Consistent Memory Models

Very different hardware memory models

None match the Java Memory Model

The JVM bridges the gap -

- While keeping normal loads & stores fast
- Via combinations of fences, code scheduling, placement of locks & CAS ops
- Requires close cooperation from the JITs
- Requires detailed hardware knowledge

Illusion: Consistent Thread Models

Very different OS thread models

- Linux/BSD, Windows, Embedded
- But also cell phones, iPad, etc

Java just does 'new Thread'

- On micro devices to 1000-cpu giant machines
- and *synchronized*, *wait, notify, join,* etc, all just work

Illusion: Locks are Fast

Contended locks obviously block and must involve the OS

- (Expect fairness from the OS)

Uncontended locks are a dozen nano's or so

- Biased locking: ~2-4 clocks (when it works)
- *Very* fast user-mode locks otherwise

Highly optimized because *synchronized* is so common

Illusion: Locks are Fast

People still don't know how to program concurrently

- The 'just add locks until it works' mentality
- i.e. Lowest-common-denominator programming
- So locks became common
- So JVMs optimized them

This enabled a particular concurrent programming style

And we, as an industry, learned a lot about

concurrent programming as a result

Illusion: Quick Time Access

System.currentTimeMillis

- Called ***billions*** of times/sec in some benchmarks
- Fairly common in all large Java apps
- Real Java programs expect that:

if T1's Sys.cTM < T2's Sys.cTM

then T1 $\lll_{happens\_before}$ T2

But could not use X86's "tsc" register until 2012

- Value not coherent across CPUs
- Not consistent, e.g. slow ticking in low-power mode
- Monotonic per CPU – but not per-thread

Illusion: Quick Time Access

System.currentTimeMillis

- Switching from fastest linux gettimeofday call
  - (mostly-user-mode atomic time struct read)
  - gettimeofday gives *quality* time not *fast* time
- To a plain *load* (updated by background thread)
- Was worth 10% speed boost on key benchmark

Hypervisors like to "idealize" tsc :

- Means: uniform monotonic ticking
- Means: slows access to tsc by 100x?

Illusions We’d Like To Have

Infinite Stack

- e.g. Tail calls. Useful in some functional languages

Running-code-is-data

- e.g. Closures

'Integer' is as cheap as 'int'

- e.g. Auto-boxing optimizations

'BigInteger' is as cheap as 'int'

- e.g. Tagged integer math, silent overflow to infinite precision integers

Illusions We’d Like To Have

Atomic Multi-Address Update

- e.g. Software Transactional Memory

Fast alternative call bindings

- e.g. invokedynamic

Illusions We Think We Have

Thread priorities

- Mostly none on Linux without *root* permission
- But also relative to entire machine, not JVM
- Means a low-priority JVM with high priority threads
  - e.g. Concurrent GC threads trying to keep up
- …can starve a medium-priority JVM

Write-once-run-anywhere

- Scale matters: programs for very small or very large machines are different

Illusions We Think We Have

Finalizers are Useful

- They suck for reclaiming OS resources
  - Because no timeliness guarantees
  - Code "eventually" runs, but might be never
  - e.g. Tomcat requires a out-of-file-handles situation trigger a FullGC to reclaim finalizers to recycle OS file handles

What other out-of-OS resources situations need to trigger a GC?

Do we really want to code our programs this way?

Illusions We Think We Have

Soft, Phantom Refs are Useful

- Again using GC to manage a user resource
- e.g. Use GC to manage Caches

Low memory causes

rapid GC cycles causes

soft refs to flush causes

caches to empty causes

more cache misses causes

more application work causes

more allocation causes

rapid GC cycles

What Is In a JVM?

A whole lot!

The V in Virtual is a **powerful** abstraction

- The JVM Provides **Illusions**
- Programmers focus on value-add elsewhere

Part 3: Beyond the JVM:
Design Choices When Making a Virtual Machine