JVM implementation challenges:
Why the future is hard but worth it

John Rose, Java VM Architect

JFokus, Stockholm, February 2015
http://cr.openjdk.java.net/~jrose/pres/
201502-JVMChallenges.pdf

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Purposes of this talk

http://cr.openjdk.java.net/~jrose/pres/201502-JVMChallenges.pdf

- Speak about JVM architecture, especially its futures
- Propose key goals (idealized, a bit) for the JVM design
- Point out obstacles that thwart those goals
- Show how the OpenJDK is making progress on the goals
- Stop talking after 40 minutes

  … having teased everyone and informed no one

Audience = designers & implementors of VMs & languages
(up-stack users may find some of this talk uninteresting)

 | |

ORACLE

VM DESIGN

JVM Vision

Platform for language execution

- Java = simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, dynamic language
  - Gosling, “Java: An Overview” (1995)
- JVM = portable execution platform featuring uniform objects, native threads, interpreted/compiled execution (a.k.a. “mixed-mode”), profile-driven speculative optimization with deoptimization — for Java
  - Sun, “The Java HotSpot Virtual Machine” (2001)

ORACLE

JVM Vision

Not just for Java

- "Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine."
  - *JVM Specification* First Edition (1997), preface
- "In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages."
- "The Java SE 7 platform in 2011 made good on [this] promise."
  - *JVM Specification*, J2SE 7 Edition (2013)
- In SE 8, Java used "polyglot" invokedynamic to implement closures.

ORACLE

What's in a JVM?

Data & code, safe & portable

- Data structures connected by managed pointers, dynamically typed
  - Computation with primitives and objects (methods, classes, interfaces)
- (Byte-)code that runs fast (hardware speed) without preprocessing
  - Name binding and optimization are deferred; lazy load and lazy link
- Safe, type-enforcing, robust, secure
  - Limits damage caused by error or malice, bug-resistant
- Portable, architecture-neutral, multiprocessing, large memory
  - Keeps pace with hardware technologies, grows with data paths & memory

 | |

ORACLE

Clever JVM moves

- Data: GC, uniform reflectability, primitives with optional boxing
- Code: JIT compilation, mixed-mode execution; profiling, deoptimization
- Safe: Redundant checks (verifier); abnormalities throw exceptions
- Portable: Clear specifications, hidden details; long-term compatibility

- The best moves are invisible, or can safely be neglected by the user.

Result: A simple user experience despite the complex technology.

Reality check

(or, how do we know when we win?)

- Data: Easy to understand, flexible, regular, compact (in memory)
- Code: Efficiently encodes source code meaning, runs fast
- Safe: Doesn't crash, hard to crack, no need for "snake pit" maps
- Portable: Gets the best out of every major CPU and system

ORACLE

Reality check

(searching and fearless inventory, anyone?)

- Data? Diffuse (too many indirections); racy; weak genericity (hard to tune)
- Code? Compilation unit size is too small; 16-bit limits; Java-specific
- Safe? The JVM is complex (~1M LOC)
- Portable? 32-bit array size; 64-bit primitive size; dinosaur threads

“By seeking and blundering we learn.” — Goethe

τὸν πάθει μάθος θέντα — “Suffering makes learning.” — Aeschylus

ORACLE

Language suffering makes for VM learning

- Language pain points show us where JVM semantics…

  … align too rigidly with Java language semantics,

  … fail to align closely with modern hardware,

  … or impose excessive costs in some other way.

- The JVM gains power and applicability by removing pain points
  - Improve simplicity and performance for new users
  - Retain compatibility and performance for present users

JVM pain points (from “Evolving the JVM”, [JVMLS 2014])

Pain Point	Tools & Workarounds	Upgrade Possibilities
Names (method, type)	mangling to Java identifiers	unicode IDs ✓1.5/JSR-202, structured names
Invocation (mode, linkage)	reflection, intf. adapters	indy/MH/CS ✓1.7/JSR-292, tail-calls, basic blocks
Type definition	static gen., class loaders	specialization, value types
Application loading	JARs & classes, JIT compiler	Jigsaw, AOT compilation
Concurrency	threads, synchronized	Streams ✓1.8/JSR-335, Sumatra (GPU), fibers
(Im-)Mutability	final fields, array encap.	VarHandles, JMM, frozen data
Data layout	objects, arrays	Arrays 2.0, value types, FFI
Native code libraries	JNI	Panama

+ sun.misc.Unsafe

ORACLE

BIG GOALS

What should the JVM look like in 15 years?

(eight not-so-modest goals)

- Uniform model: Objects, arrays, values, types, methods "feel similar"
- Memory efficient: tunable data (cf. C/C++), naturally local, pointer-thrifty
- Optimizing: Shared code mechanically customized to each hot path
- Post-threaded: Routine confinement/immutability, natural concurrency
- Interoperable: Robust integration with non-managed languages
- Broadly useful: Safely and reliably runs most modern languages.
- Compatible: Runs 30-year-old dusty JARs.
- Performant: Gets the most out of major CPUs and systems.

CHALLENGES

ORACLE

Problem: cache is scarce, memory is far away

what Moore's Law didn't tell you

- Rule #1: Cache lines should contain 50% of each bit (1/0)
  - E.g., if cache lines are 75% zeroes, your D$ size is effectively halved
- Rule #2: Use a cache-line worth of data per random access
  - Chaining through a 8-byte pointer could waste 75% of a cache line fetch
- The ideal: Make sequential accesses to high-entropy payloads
- Today's JVM violates both rules by its heavy reliance on pointers
  - Arrays can help, but they often devolve to *array-of-pointer-to-payload*
- The JVM can't express one value containing another (except via pointer)

Problem: sharing is hard

The dark side of late binding

- Every JVM starts by rebuilding key code and data from scratch
  - Would like to time-shift repeated steps and share the results
  - Or, would like to checkpoint the first time, and reuse it
- Within a JVM, some data (such as arrays) must be defensively copied
  - Would like a way to reuse sharable data, without worrying about mutation
  - Array constant data should be allocated in a shared library, not in the heap
- In general, sharing failures appear as excess footprint (at various scales)
  - We can detect the repetitions, when we are lucky, but not always

 | |

ORACLE

Problem: hot-path customization

(or, loop versioning for objects)

- Generic shared code (like Arrays.sort) suffers from “profile pollution”
  - It’s loaded once, but needs to do very different tasks for different inputs
  - The inner loop comparison (Comparator.compare) is megamorphic
  - Making a new copy for each use allows full optimization
  - Key technique: JIT a customized copy for each distinct “hot” use
- Challenge #1: Detect hot uses that can be split off and optimized
- Challenge #2: Don’t do this too often, since the JIT is expensive
- Ex: code cache for Arrays.sort keyed on two types (a.class, c.class)

Syndrome-driven customization

(better algorithms through argument science)

- Generic algorithms are sensitive to their arguments
  - Both argument type and structure (array length, stream pipeline)
- Today, optimization follows upstream type profiling with heavy inlining
  - This breaks down if the upstream profile is in another thread (FJ on streams)
- Tomorrow's customization needs to detect repeated "syndromes"
  - Syndrome is relevant type and structure of sensitive operands
  - A hot syndrome needs a customized code version, fully optimized
  - A generic call site needs a "switch" to route the call to its code version

ORACLE

Examples of “syndrome” based customization

- In standard profiling, the receiver’s exact type is the syndrome
  - Exact types allow downstream devirtualization and inlining
  - You have pretty much won, if you can inline everything in a loop
- Hidden classes or maps (Self, JavaScript, JSR 292)
- A stream pipeline (minus the source) is probably a syndrome
  - Execute the same loop kernel, across a FJ pool
  - Recognize the use of the loop kernel, and compile it just once
- Syndromes work like classes, except they are hidden and emergent

ORACLE

Problem: threads are passé

- Java distinguished itself 15 years ago with a big investment in threads
  - At that time threads were hard to program; memory models were unreliable
- Threads use enormous amounts of stack memory
  - Their built-in synchronization mechanisms are obsolete
  - They cannot do SIMD lockstep synchronization, as today's GPUs require
  - A thread cannot release its resources until its initial task is done
- Event-driven or reactive code decomposes into many concurrent tasks
  - Would be overkill to give each task a thread

ORACLE

Beyond threads: fibers, warps, events, reaction

- A "fiber" is the lively bit of a thread, minus the large control stack
  - Fibers run on host threads, treating them as interchangeable commodities
  - It is reasonable to fire up a million fibers and set them running
  - A fiber might be an array, an index, and a bit of code to run for that index
- A "warp" is a group of similarly-shaped fibers which advance together
  - A warp processing an array could have fibers differing only by array index
- Fibers can wait on events, without tying up thread resources
- Reactive programming combines fibers to process events

Beyond threads: VM support for fibers etc.

- Fibers work best when three features coincide:
  - It is easy to start a fiber from a **closure** (anonymous function plus data)
  - An unfinished fiber execution can be paused, yielding a **continuation**
  - A finished fiber invoke a chosen successor using a **tail-call**
- All of these features require additional JVM work to run smoothly
- When a fiber is created, it should package its continuation into heap-frames
  - It should be possible to teach the JIT to run a heap-frame directly
- JNI needs “suspend/resume hooks” to work properly with fibers

ORACLE

Beyond threads: varying the variables

- Java threads communicate by reading and writing heap variables
  - Complicated rules determine whether there are “races”
- Variables used for thread communication are very hard to tame
  - As threads go light, the taming becomes more complicated
- We need better design patterns so variables are safe to use
  - Frozen (im-mutable) variables are safe to use by any thread
  - Variables protected by a lock are confined to the locking thread
  - Changeable data structures should be proven race-safe
- We need fewer variables which are vulnerable to races

ORACLE

Problem: universality

Can one VM rule them all?

- Dynamic languages have a wide variety of object models
  - Lisp, Ruby, Python, JavaScript
  - Executing them well requires a cheap simulation with JVM objects
- Statically typed languages
  - Requires cheap mapping of static types to JVM types
  - Parameterized types and value types may help in many cases
- JVM must bootstrap language runtimes quickly (see “sharing”)

Towards universality

Can a VM make some new friends?

- CPUs now operate on 256-bit (soon 512-bit) data
  - JVM needs flat value types of those sizes (Valhalla value types will do it)
- Java-specific linkage paths need generalizing
  - JVM could support ops like “getfield” with some programmability
  - Ex: If a getfield fails to link, ask the link-ee to construct an accessor to run
- Assign each array type an interface, so clients can virtualize arrays
  - This way library-defined arrays and standard arrays can interoperate

Problem: interoperability

Can one VM connect to them all?

- Non-managed languages cannot be trusted to touch managed pointers
  - (The JVM uses “handles” to stand in for pointers, via JNI)
- Meanwhile, Java has a hard time working with some foreign calls
  - Vectors larger than 64 bits are hard to represent
  - APIs export data structure layouts (struct stat) which are hard to traverse
  - The “cultural practices” (like safety) are different between Java and C

 | |

ORACLE

Aspiring to interoperability

- Header files can be parsed by Java tools (the “jextract” tool)
  - The output is Java-centric metadata, interfaces that express the API ops
  - An improved JNI binder can connect these interfaces directly to C APIs
- Value types can represent exotic C/C++ values, or even T* pointers
  - Type parameter specialization can model many of the types C needs
- The raw imports will be unsafe (almost always)
  - Wrappers will need to be engineered to upgrade the safety of the API

Problem: compounding complexity

- OpenJDK HotSpot consists of about 900k lines of C/C++ code
- Original Mercurial check-in (2007) was about 600k LOC
  - About 450k LOC have not changed since then
- Projection: In another 15 years this process will iterate twice.
  - 1900k LOC in 2030 JVM, containing 250k LOC of 2007-vintage
- Some modules seem to be near a “complexity ceiling” (hard to change)
  - Optimizing JIT, object tracing code in GC, dynamic linkage code paths
- Meanwhile, C++ is growing less compatible with Java (heap management)

ORACLE

Combating complexity

- Culture of clean: Leave it nicer than you found it
  - Budget for technical debt, expect and welcome debt removal
- Java-on-Java: Get metacircular, code more in fewer lines
  - Existence proofs: Jikes, Maxine, Graal, Substrate VM, method handles
  - To make practical, requires investment in AOT or other sharing tech.
  - Bootstrapping requires late-binding decisions to be shifted earlier in time
- Better testing: Can improve code faster if errors are removed earlier
  - Unit testing requires excellent modularity (see “Java-on-Java”)
  - Exhaustive functional testing requires metaprogramming (Java!)

ORACLE

Problem: long-term compatibility

- Java keeps customers when it doesn't break their code (*ceteris paribus*)
  - As time goes on, compatibility requests become more… lengthy
  - It is difficult to retire features after releasing them
- APIs must operate reasonably across separate compilation
  - Old clients of updated APIs must get reasonable behavior
  - Also must support old subclass of new superclass, and vice versa
- You can never change the meaning of a name, even if it's bad
  - Ex: Identity semantics (& public constructor) of java.lang.Integer

ORACLE

Problem: long-term compatibility
(a few of the names we want to change)

- StringBuffer was synchronized; had to replace it by StringBuilder
  - Likewise Vector replaced by ArrayList
- In 1.1 we needed to fix a case of method name resolution
  - Invented the ACC_SUPER bit, to mark classes with the bug fixed
  - Today, the JVM requires the presence of the ACC_SUPER bit
- future problem: object identity
  - We can treat a java.lang.Integer as a new-style boxed int
  - But only if we can prove no-one tries to observe its object identity

ORACLE

Prolonging compatibility

- VM support for adding API points (default methods in SE 8)
- VM support for deleting API points
  - Conditionally-present methods? Multiple versions of one class?
- Simultaneous versions may assist with some code-sharing problems
  - Ex: Extension methods allow users to share and tweak at the same time
- Need VM & tooling assist to audit program usage of stale APIs
  - Some of this work can be done on class-files, as with FindBugs
  - VM-level deprecation (warn/replace/error?)

OPENJDK

OpenJDK Project Valhalla: Specialized generics

- We are experimenting with true parametric polymorphism.
  - List<int> is not sugar for List<Integer>; ArrayList<int> has a real int[] array
  - List<?> is willing to operate on primitives as well as erased references
- The design issues are subtle, especially to preserve compatibility
- The customization problem: When/how to “split” List<int> from List<?>
- Requires a story for “reifying” type parameters (like “int”)
- May require some “universal” infrastructure, such as virtualized getfield
- May require a true “any” type, and/or new bytecodes

OpenJDK Project Valhalla: Var-Handles

- Goal: Apply a memory fence or make a volatile load or cas
  – Can do this awkwardly and unsafely with sun.misc.Unsafe
  – Can now do it safely with a construct called VarHandle (cf. MethodHandle)
- Requires some language and JVM support, as with method handles
- Appears to require some type of parametric polymorphism

OpenJDK Project Valhalla: Value Types

- This is the big goal, to make new types which operate like primitives
- Slogan: *Codes like a class, works like an int!*
  - You can structure it with fields and methods — Complex{double re, im;}
  - In the end it's a value you pass around, not an object you peek and poke
- Provides useful means to:
  - Flatten memory (embed values, not link to them by pointers)
  - Create lightweight containers (Optional<T>)
  - Build secure cursors and pointers (semantics of a checked C char**)

ORACLE

OpenJDK Project Panama: jextract

- Panama is building an experimental tool, jextract.
  - Reads header files, writes Java-centric metadata (as described above)
  - Tool uses libclang for a high-quality parser
  - Tool can successfully extract the libclang API, close to self-hosting
- Next step is to modify JVM's JNI binder to accept jextract output
  - In most cases, no adapter code is needed (C++ may require adapters)
- Java and C programmers should have about equal access to C APIs!

ORACLE

OpenJDK Project Panama: layout engine

- Foreign code is no good without foreign data
- Jextract will also emit metadata that describes structs, typedefs, etc.
- This “layout metadata” will be general beyond C/C++
  - Other “little langauges” such as XDR can be targeted to Java metadata
  - Again, no adapter code is needed in most cases
- Java and C programmers should have about equal access to layouts!

 | |

OpenJDK Project Panama: Arrays 2.0

- Building on top of the layout engine, we are defining new array APIs
- These APIs will apply to old Java arrays, using some retrofitting tricks
  - Current projects: Adding immutability (freezing) to old arrays
  - Supporting compact initialization of arrays (no more giant <clinit>)
  - Retrofitting interfaces with default methods to old array types
- Array APIs will also apply to new-style, programmed-layout arrays
- They will work well with Valhalla generics and value types.
  - Array<Complex<double>> will be a welcome type to numeric programmers

ORACLE

That’s a lot to digest!

- We agree. It will take us several releases to get to Java-2030
- Meanwhile, as Brian recently said of JVM Stewardship:
  - If we don’t know how to do it right, we won’t do it (now).
- It’s better to leave out something good than to put in something bad
  - Because compatibility is forever.
- And even if we find something good, it has to carry its weight.
  - Less is more — The first Java “buzzword” is **SIMPLE**
- So we are experimenting, and at this point we are quite hopful…

QUESTIONS?

ORACLE