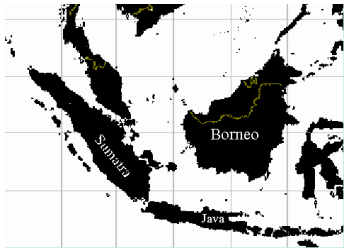
Overview of numerics on the Java Platform: past, present, and possible futures

FPBench Community Meeting

Joseph D. Darcy (OpenJDK darcy, jddarcy, @jddarcy, @jddarcy)
Member of the Technical Staff, Java Floating-point Czar emeritus
Java Platform Group, Oracle
November 2, 2023

Who am I?



- As a student:
  - UC Berkeley masters project: *Borneo 1.0: Adding IEEE 754 floating point support to Java*, 1998
  - Co-author (with W. Kahan) of *How Java's Floating-Point Hurts Everyone Everywhere*, ACM 1998 Workshop on Java for High-Performance Network Computing
- After entering industry:
  - One of the editors of the IEEE 754 2008 revision.
  - Started working in Sun's Java platform group in 2000; have worked in that group since that time, including after the Oracle acquisition in 2010. On the numerics front:
    - Expanded set of methods provided in Java's math library
    - Added support for hexadecimal floating-point literals to the platform
    - Worked on effort to add floating-point rounding to `BigDecimal`; exact divide algorithm
    - Found bug in FDLIBM pow; ported FDLIBM from C to Java
    - JEP 306: Restore Always-Strict Floating-Point Semantics
    - But mostly work on other aspects of Java, such as core libraries and reviewing changes for compatibility
  - On the Java language front, was lead engineer for Project Coin / JSR 334 in JDK 7, etc.

Outline

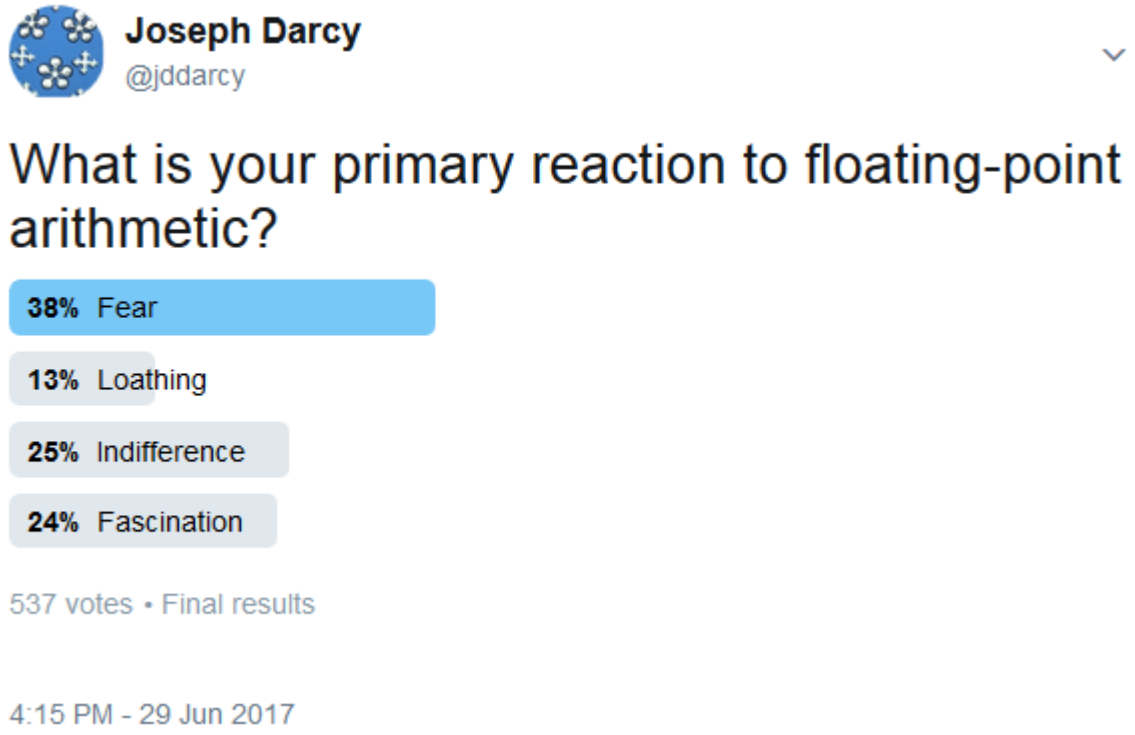
- Background
- Java Numerics: Present
- Java Numerics: Past
- Java Numerics: Possible futures
- Questions
- Slides will be available from https://github.com/jddarcy/SpeakingArchive

Background

- Assumptions about the audience:
  - Have heard of Java, may not be familiar with the details.
  - May have looked at Java a number of years ago; if so, many of the relevant details may have changed since then.
- Will give a brief overview of current Java development practices and technology, happy to discuss in more detail if there is interest.
- Relate Java numerics support to long-standing design philosophy of the platform.
- Difficult to be simultaneously concise, clear, and correct in communication; may favor concisely to avoid bogging down in less relevant details.
- Java platform has been used for over 25 years, many changes over time!

Another audience assumption

Floating-point more frightening & more loathsome than spiders, but not to FPBench



(If you've already overcome arachnophobia, level-up to take on tailless whip scorpions!)

What is the Java platform for the purposes of this talk?

- Specifically for Java SE (standard edition), primarily servers and desktops
- Essential specifications
  - The Java Programming Language
  - The Java Virtual Machine
  - Core libraries
- Implementations of the above in OpenJDK
- Will focus comments on mainline OpenJDK implementation; other implementations can and do vary, variations over time
  - JDK 21 is the most recent feature release to ship in September 2023
  - For the last several years, the JDK has had two feature releases per year, one in March and one in September.

Buzzword compliant description of Java, circa 1995

"Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language."

The Java™ Language: An Overview, Sun Microsystems 1994, 1995

Java Numerics: Present

Overview of Java numerics today, JLS and JVMS

- In the Java programming language:
  - 32-bit `float` type, corresponding to IEEE 754 2019 binary32
  - 64-bit `double` type, corresponding to IEEE 754 2019 binary64
  - {+, -, *, /} operations; expressions *must* be evaluated as-if as written – in other words no hidden promotion of `float` intermediates to `double`, no implicit replacement of `(a*b + c)` with `fusedMultiplyAdd(a, b, c)`.
  - Single and multi-dimensional `float` and `double` arrays
  - Correctly-rounded binary ↔ decimal conversion
- In the Java Virtual machine
  - 32-bit `float` and 64-bit `double` types, including arrays
  - `fadd`, `fsub`, `fmul`, `fdiv`, `dadd`, `dsub`, `dmul`, `ddiv` instructions
    - VM *cannot* make transformations that would alter the computed result, fma, etc.

Overview of Java numerics today, core libraries

- Two side-by-side math libraries with the same set of methods: `java.lang.Math` and `java.lang.StrictMath`:
  - `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `sqrt`, `abs`, `pow`, `exp`, `log`, `log10`, `expm1`, `log1p`, `fma`, IEEE recommended functions, …
  - `Math` and `StrictMath` have the same quality of implementation criteria (some number of ulps and semi-monotonicity)
  - For cross-platform and cross-release reproducibility, `StrictMath` is specified to use the FDLIBM algorithms
    (HT Paul Zimmerman, Jean-Michel Muller, and others who have worked on correctly rounded math libraries, these weren't seen as mature enough to use at the time.)
  - For (possibly) platform-optimized implementations, use `Math` instead. Many `Math` methods have VM *intrinsics*.

Overview of Java numerics today, core libraries, cont.

- Bitwise conversion between same-sized integral and floating-point types
- Basic support for `float` ↔ float16 conversions
- New binary → decimal base conversion algorithm from Raffaello Giulietti

Overview of Java numerics today, core libraries, cont.

- `java.math.BigInteger`
  - Arbitrary precision integer arithmetic
  - Basic arithmetic and logical operations (method calls, not operator syntax); primality testing, etc.
- `java.math.BigDecimal`
  - Arbitrary precision decimal arithmetic with fixed-point *and* floating-point style rounding operations
  - Basic arithmetic operations (method calls, not operator syntax) and square root
  - Predates IEEE 754 support for decimal, various differences from 754 standard (no infinite or NaN values, only positive zero)

Other constructs

- Besides object-oriented features in the language, rich assortment of imperative programming structures, `for`-loops and `while`-loops, etc.
- Arrays with 32-bit indexes
- Multi-dimensions arrays may be “ragged” – arrays of arrays with the component arrays being separate objects.

With these capabilities, determined programmers can write FORTRAN in Java too.

Multiple programming paradigms possible and supported on the platform, object-oriented, imperative, ….

In progress, vector support

- *Incubating* API `jdk.incubator.vector`:
  "Defines an API for expressing computations that can be reliably compiled at runtime into SIMD instructions, such as AVX instructions on x64, and NEON instructions on AArch64."
  - Abstraction more at the level of vector registers than mathematical vectors; in particular, the user has to write the strip-mining code.
  - Anticipated to be added as a normal feature once Project Valhalla allows value types to be declared

Facilities *not* included in the JDK as of 2023

- Matrix/vector/tensor linear algebra libraries, including BLAS
- Visualization (along the lines of gnuplot or Mathematica)
- Statistical analyses, other than those found in `java.util.stream`
- Complex numbers ($x + yi$)
- Full support for other “simple” numerical types like float16, bfloat16, …
  - Adding more numerical types that act similarly to built-in types on their memory behavior and performance characteristics also gated on Project Valhalla.
  - Keeping an eye on IEEE P3109: Standard for Arithmetic Formats for Machine Learning
- (Java libraries doing these computations can be found outside of the JDK; can use JNI and other protocols to call into C (and other native) libraries.)

Also *not* included

- Built-in support for directed rounding of `float`/`double` operations (could implement *slowly* by converting `float`/`double` operands to `BigDecimal`, doing an exact operation in `BigDecimal`, and directed rounding back to `float`/`double`.)
- Support for IEEE 754 sticky flags or alternate exception handling

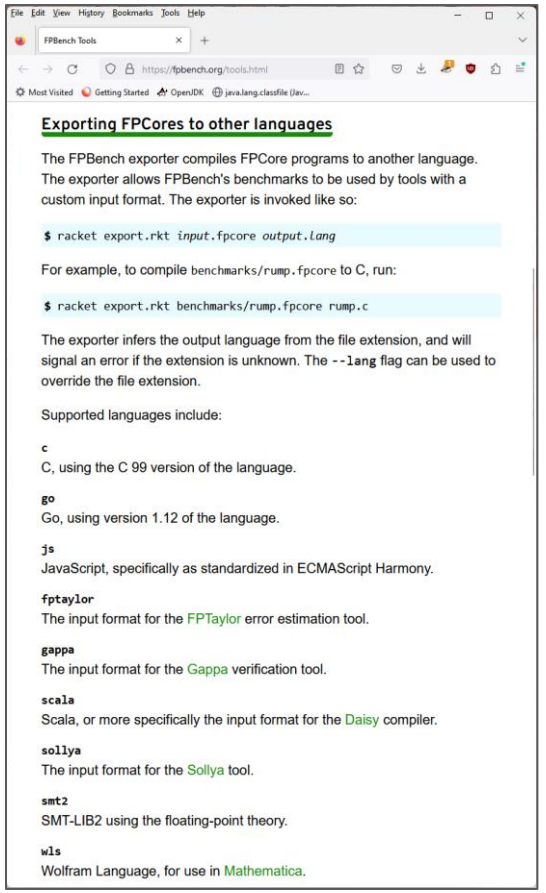
Benchmarks

One measure of performance today

Don’t see Java as a supported language for FPBench Tools…

..yet :-)

- I assume the Scala/Daisy code is run through a JVM backend.
- If start looking into Java benchmarks, recommend using jmh (Java Microbenchmark Harness) to help get reliable results given the complexities of JVM startup, etc.



Exporting FPCores to other languages

The FPBench exporter compiles FPCore programs to another language. The exporter allows FPBench's benchmarks to be used by tools with a custom input format. The exporter is invoked like so:

```
$ racket export.rkt input.fpcore output.lang
```

For example, to compile `benchmarks/rump.fpcore` to C, run:

```
$ racket export.rkt benchmarks/rump.fpcore rump.c
```

The exporter infers the output language from the file extension, and will signal an error if the extension is unknown. The `--lang` flag can be used to override the file extension.

Supported languages include:

c
C, using the C 99 version of the language.

go
Go, using version 1.12 of the language.

js
JavaScript, specifically as standardized in ECMAScript Harmony.

fptaylor
The input format for the FPTaylor error estimation tool.

gappa
The input format for the Gappa verification tool.

scala
Scala, or more specifically the input format for the Daisy compiler.

sollya
The input format for the Sollya tool.

smt2
SMT-LIB2 using the floating-point theory.

wls
Wolfram Language, for use in Mathematica.

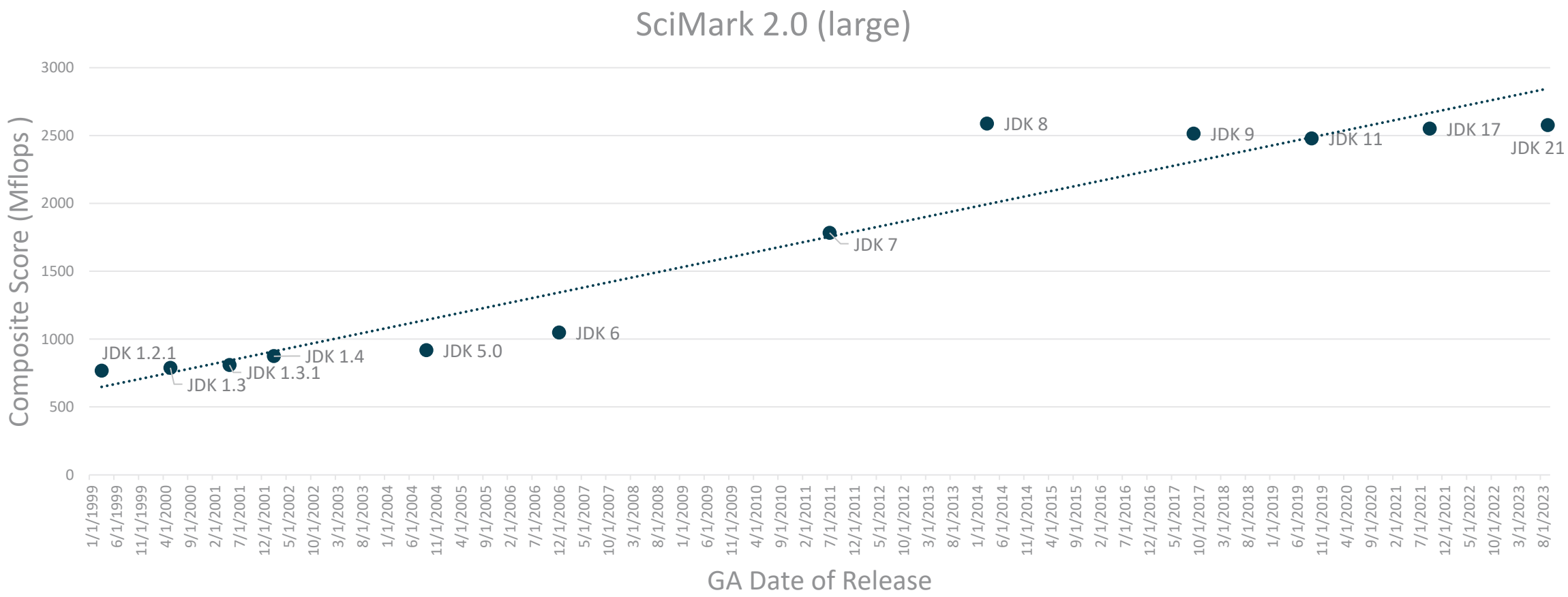
SciMark 2.0 Measurement Methodology

- On the same laptop running 64-bit Windows 10 Pro
  - Processor Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz   2.59 GHz
  - Installed RAM 32.0 GB (31.8 GB usable)
- Run `$JDK/bin/java.exe –jar -classpath scimark2lib.jar jnt.scimark2.commandline –large`
  - The jar file from SciMark website, compiled with `javac` from Sun's JDK 1.2.
  - First run to handle any file system caching effects discarded
  - Report average of next 10 runs
  - For selected releases, include "`-server`" JVM flag; otherwise, *no* tuning
  - Use 64-bit JDK when a release has one
  - Run JDKs from 1.2.1 to 21; covers more than two decades of JDK releases
    - Not trying to include all releases

SciMark2.0a components; sample score (Mflops)

- SciMark 2.0a
  - Composite Score: 2569.3041241601713
  - FFT (1048576): 289.5559167821872
  - SOR (1000x1000): 1466.9182765103376
  - Monte Carlo : 1032.9406695239807
  - Sparse matmult (N=100000, nz=1000000): 1740.0169246799883
  - LU (1000x1000): 8317.088833304362
- SciMark components incorporated into SPECjvm2008

SciMark 2.0 Scores, same laptop, varying JDK



SciMark Analysis

- (Benchmark single-threaded.)
- 20+ year old artifacts still work!
- Significant JVM improvements over time
  - Shows benefit of using class files as a level of indirection
  - Allows platform-optimized code to be used at runtime
- Particular improvements:
  - More flexible specification for `java.lang.Math` methods starting from JDK 1.3
  - Switch from using legacy x87 FPU instructions to SSE${N}, N ≥ 2, AVX{$K}?, etc.
  - Many other JVM optimizations, including possibly autovectorization

How is Java developed today?

JCP model

The Java Community Process, starting circa 1998

JCP Triad for the Java SE 21 Umbrella JSR 396

Specification	Reference Implementation (RI)	Technology Compatibility Kit (TCK)

JLS
JVMS
`java.*`
`javax.*`
…
	Build of OpenJDK 21
	On Linux and Windows
	https://jdk.java.net/java-se-ri/21
		JCK 21

JCP and Open Source

- Since JCP 2.5 in 2002, the JCP has allowed and embraced open source development of the technologies standardized through the process
- For Java SE, the *reference implementation* is built from sources in OpenJDK repositories

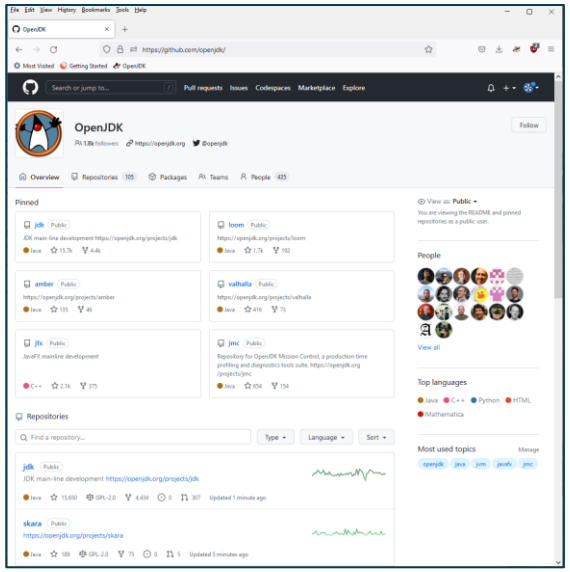
OpenJDK community and development in the open

- Good faith down payment from Sun by publication of HotSpot and `javac` sources in 2006; rest of JDK in 2007 (longer effort to remove all binary plugs, etc.)
- OpenJDK is licensed under open source licenses, predominantly GPLv2, (with the ClassPath Exception for the libraries)
- Initial OpenJDK sources populated from the in-progress JDK 7.
- A "backward branch" from JDK 7 used to make OpenJDK 6.

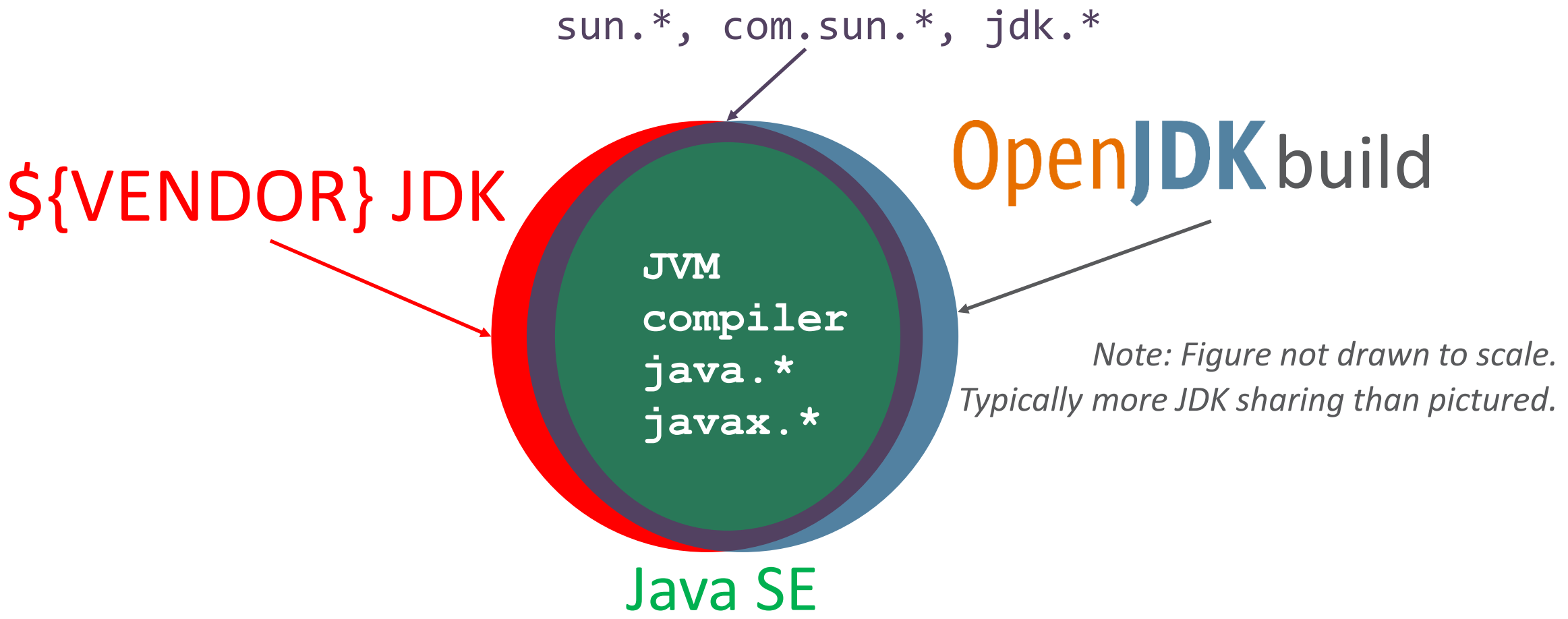
General OpenJDK infrastructure

Including, but not limited to:

- Mailing lists; used for different technology areas (core-libs, HotSpot, client-libs, etc.) and well as various projects (Amber, Valhalla, etc.)
- JBS – JDK Bug System: Jira instance, most bugs of interest are in the “JDK” project
- OpenJDK Wiki
- OpenJDK Project on GitHub



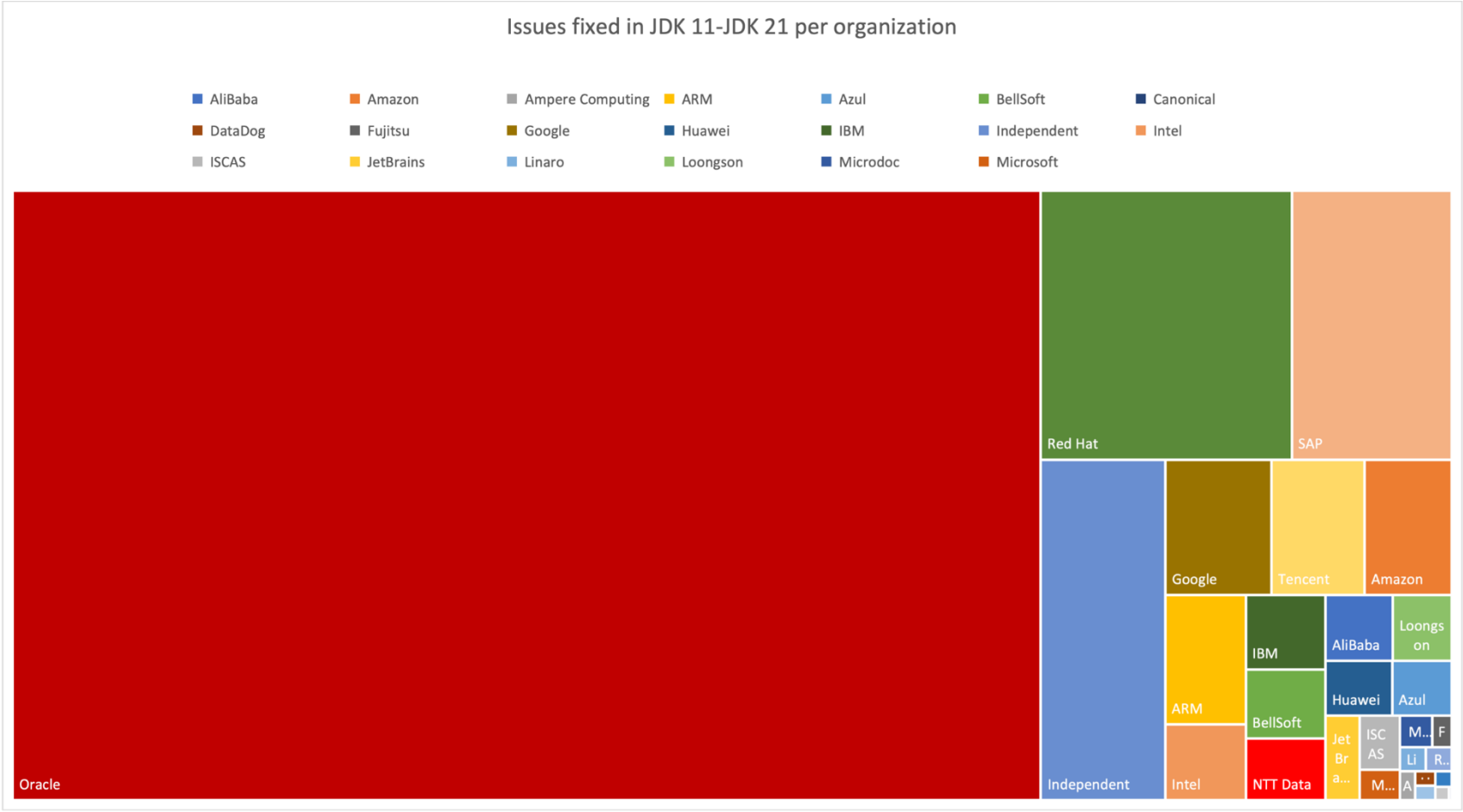
${VENDOR} JDK & OpenJDK builds, Java SE interfaces



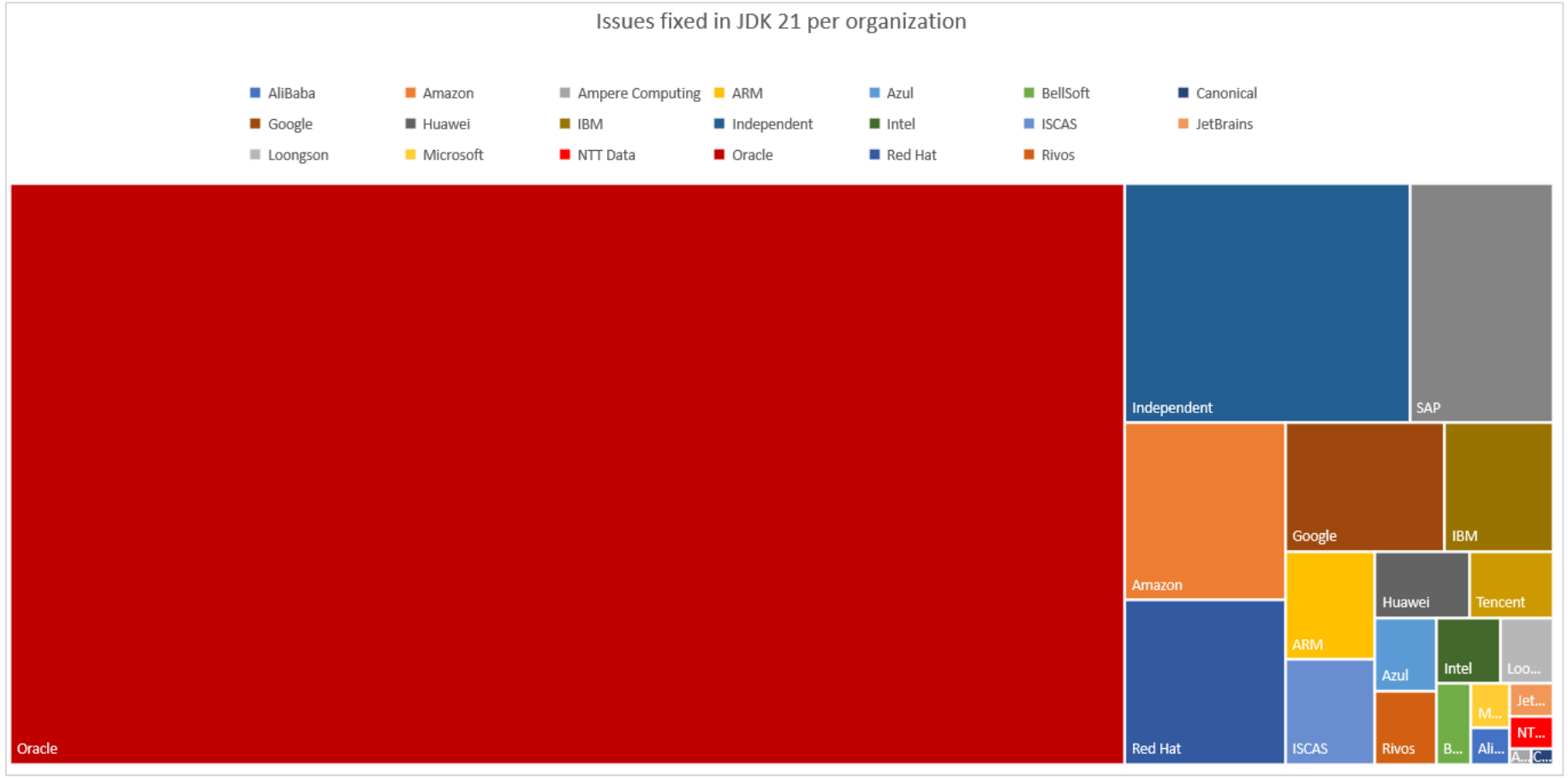
Note: Figure not drawn to scale.
Typically more JDK sharing than pictured.

Who has done what, JDK 11 to 21, inclusive

The Arrival of Java 21! by Sharat Chander



Who has done what, most recently, in JDK 21



Terminology note: “compiler”

At least two common meanings in the JDK context

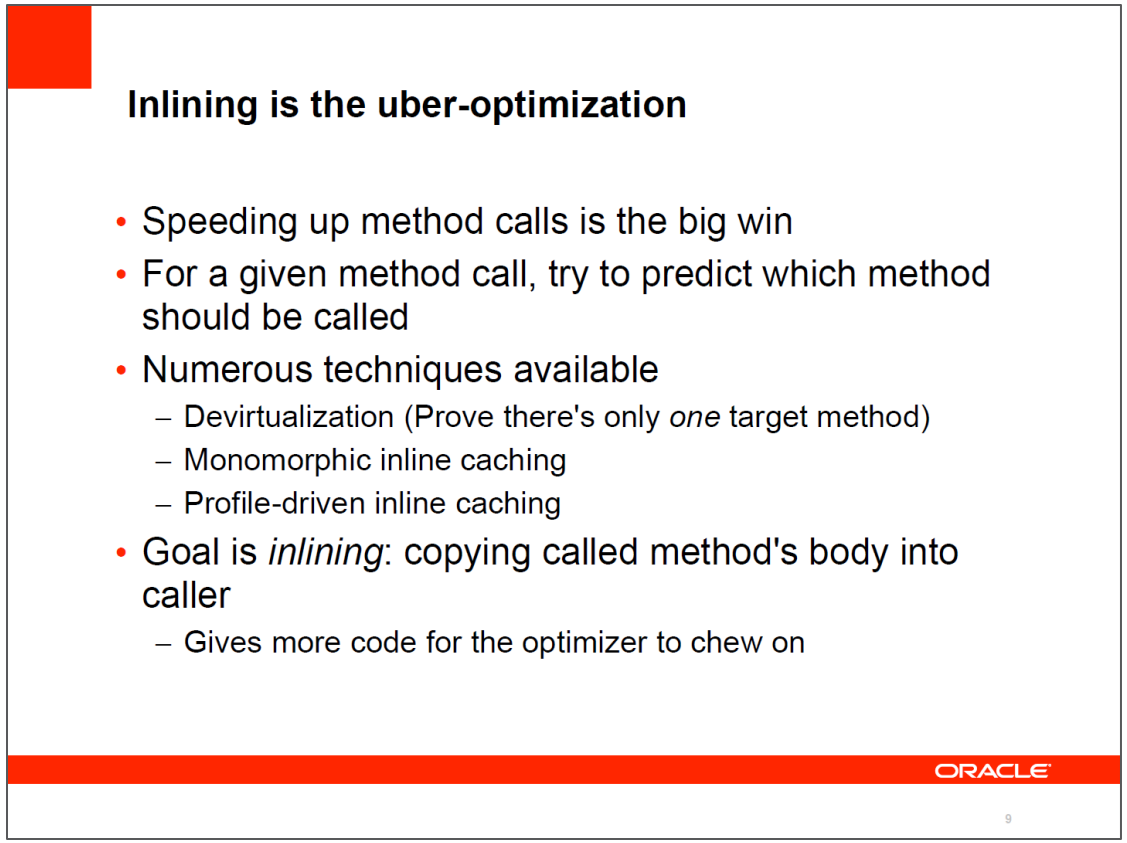
- Can refer to either:
  - Source code to class file translation
  - Class file to machine code translation inside a JVM by a JVM subsystem

Overview of HotSpot capabilities and design

- Compiler (class files → machine code), garbage collection (more broadly memory management), and runtime (RTTI, etc.) subsystems.
- Initially a class file runs under an interpreter, if it is run often enough it gets compiled to collect more detailed profiling data, if it gets run more often still, gets recompiled subject to more aggressive optimization.
- Inlining is a key optimization

Excerpts from *One VM, Many Languages*,

Rose & Goetz, JavaOne 2010



Excerpts from *One VM, Many Languages*,

Rose & Goetz, JavaOne 2010

JVM Engineering tradeoffs

- Code generation, etc. running concurrently with the application
- Different performance pressures than an off-line compilation like gcc or LLVM
- JVMs generally do some amount of profile guided optimization at runtime

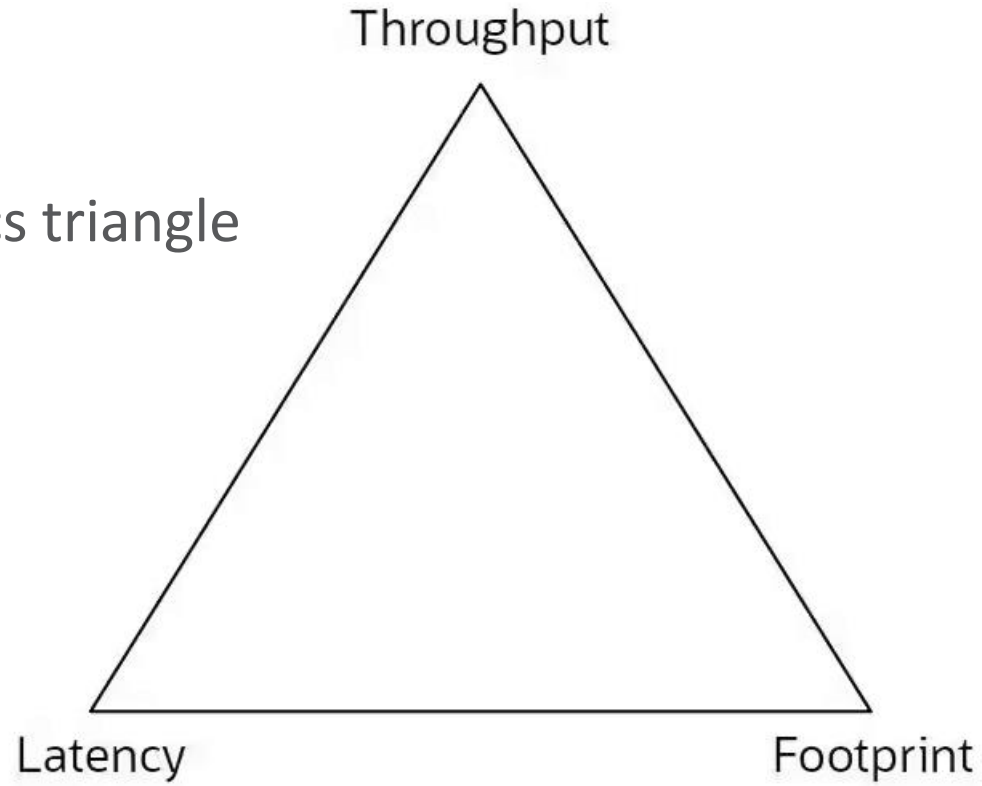
Memory Management

- Numerical computations don’t just use FLOPS, use memory too.
- Memory management in Java involves *garbage collection* (GC)
  - Best description I’ve heard for the benefits of GC: it turns a global question “will anyone need this memory/object again for the entire life of the program?” into a local question “do I need to keep a pointer to this memory/object anymore right now?”
  - GC and the memory management system do need significant engineering to run well under a range of workloads

Java garbage collection: The 10-release evolution from JDK 8 to JDK 18 by Thomas Schatzl

Next few slides derived from this blog entry

The GC performance metrics triangle



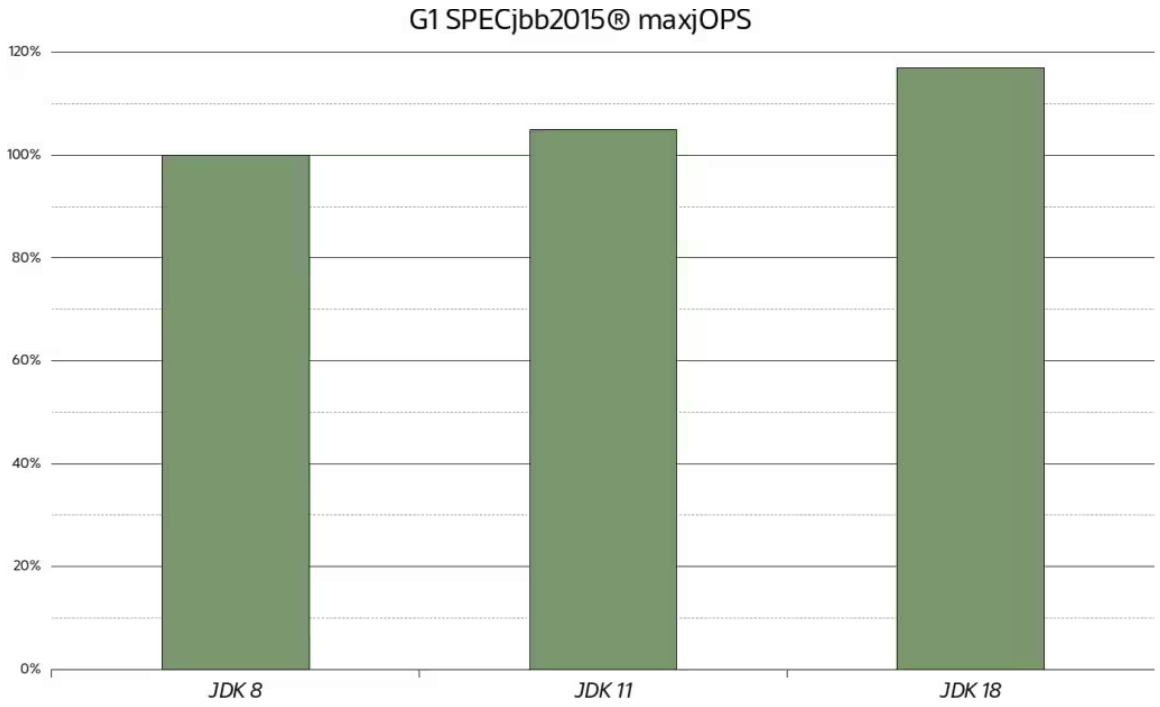
Overview of OpenJDK GC algorithms, from *10-release*…

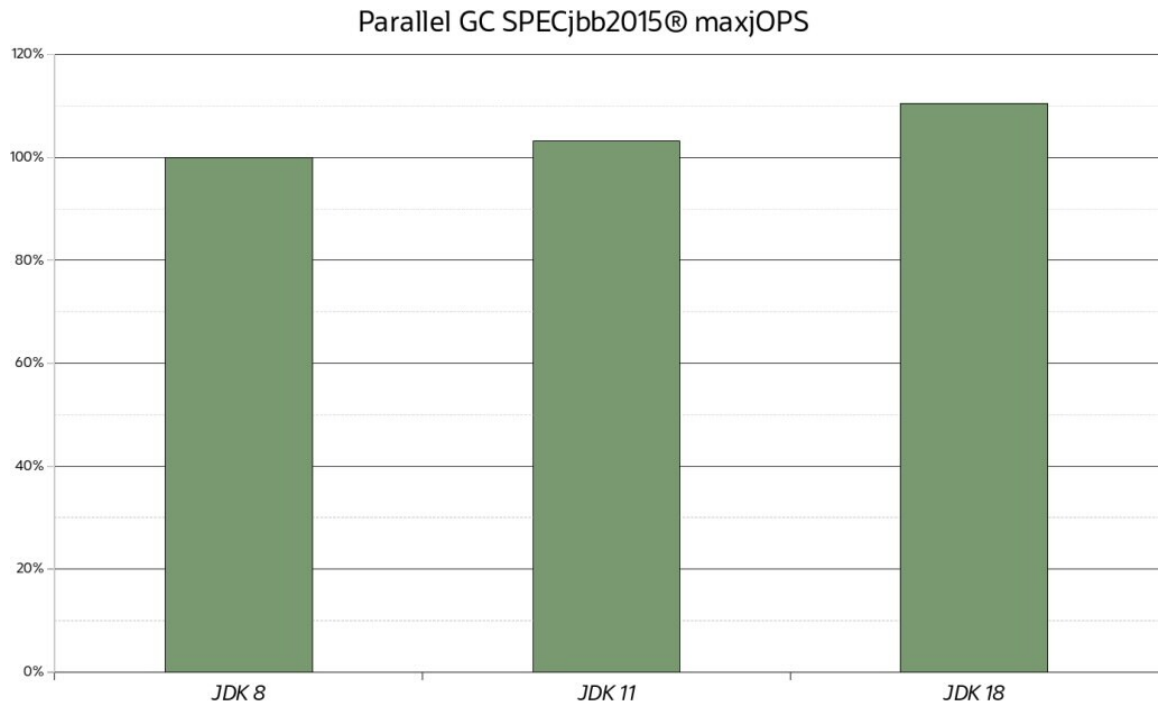
Garbage Collector	Focus Area	Concepts
Parallel	Throughput	Multithreaded stop-the-world (STW) compaction and generational collection
Garbage First (G1)	Balanced performance	Multithreaded STW compaction, concurrent liveness, and generational collection
Z Garbage Collector (ZGC)	Latency	Everything concurrent to the application
Shenandoah	Latency	Everything concurrent to the application
Serial	Footprint and startup time	Single-threaded STW compaction and generational collection

- (Generational ZGC added in JDK 21)

Throughput benefits of upgrading, from *10-release*

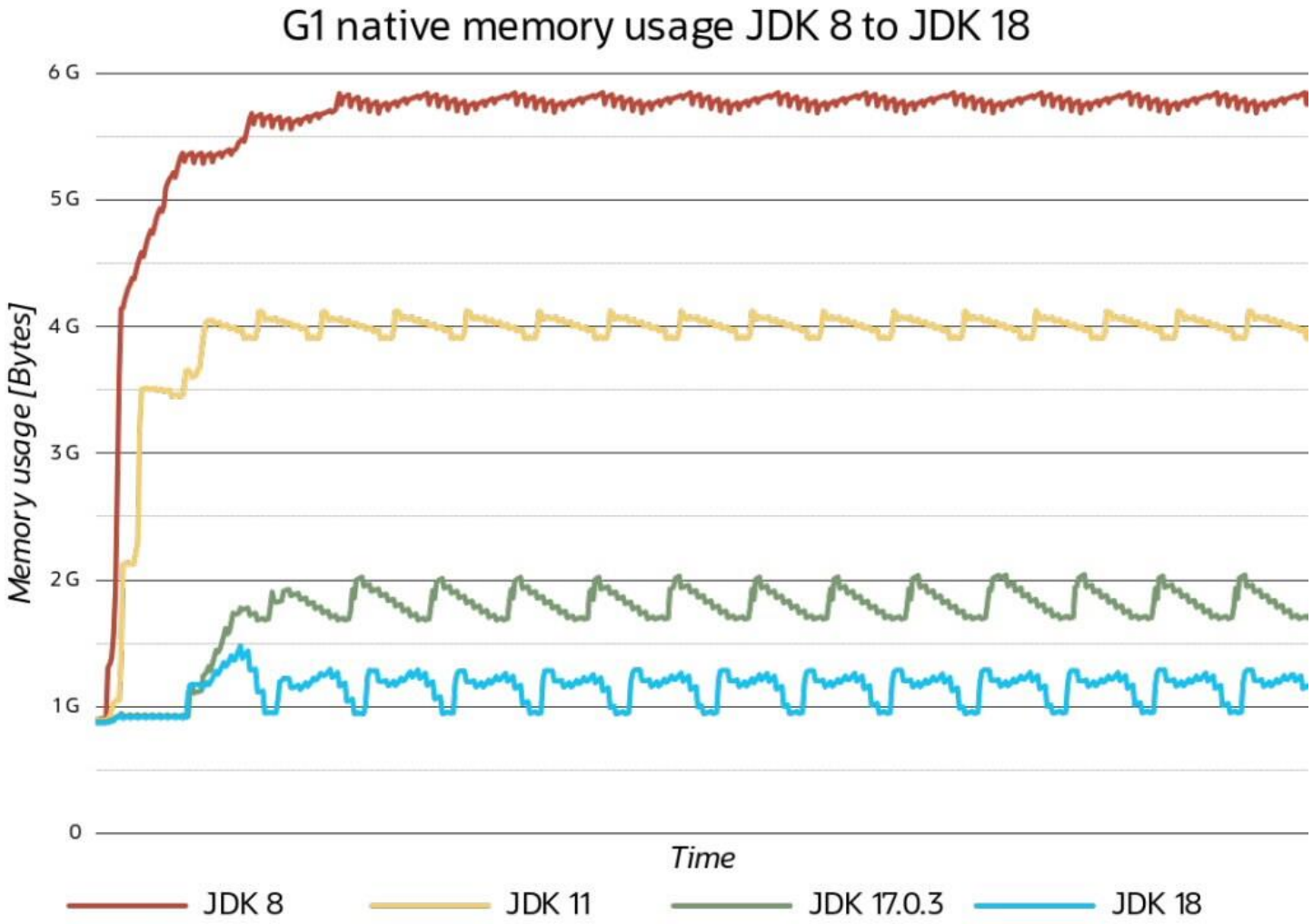
(higher is better)





G1 native memory usage, *10-release*

(lower is better)



But wait, there (can be) more metrics of interest

From "Modern garbage collection: A look at the Go GC strategy" by Mike Hearn

"Here are the different factors you will want to think about when designing a garbage collection algorithm:

- **Program throughput**: how much does your algorithm slow the program down? This is sometimes expressed as a percentage of CPU time spent doing collection vs useful work.
- **GC throughput:** how much garbage can the collector clear given a fixed amount of CPU time?
- **Heap overhead:** how much additional memory over the theoretical minimum does your collector require? If your algorithm allocates temporary structures whilst collecting, does that make memory usage of your program very spiky?
- **Pause times:** how long does your collector stop the world for?
- **Pause frequency:** how often does your collector stop the world?
- **Pause distribution:** do you typically have very short pauses but sometimes have very long pauses? Or do you prefer pauses to be a bit longer but consistent?
- **Allocation performance:** is allocation of new memory fast, slow, or unpredictable?
- **Compaction:** does your collector ever report an out-of-memory (OOM) error even if there's sufficient free space to satisfy a request, because that space has become scattered over the heap in small chunks? If it doesn't you may find your program slows down and eventually dies, even if it actually had enough memory to continue.
- **Concurrency:** how well does your collector use multi-core machines?
- **Scaling:** how well does your collector work as heaps get larger?
- **Tuning:** how complicated is the configuration of your collector, out of the box and to obtain optimal performance?
- **Warmup time:** does your algorithm self-adjust based on measured behaviour and if so, how long does it take to become optimal?
- **Page release:** does your algorithm ever release unused memory back to the OS? If so, when?
- **Portability:** does your GC work on CPU architectures that provide weaker memory consistency guarantees than x86?
- **Compatibility:** what languages and compilers does your collector work with? Can it be run with languages that weren't designed for GC, like C++? Does it require compiler modifications? And if so, does changing GC algorithm require recompiling all your program and dependencies?"

Java Numerics: Past

Platform Design Philosophy

Starting aside: Where to put the levels of indirection?

- One key place – the class file
  - Allows JVM to optimize for the processor a program is running on
  - Can be used to convey the interfaces of a library during compilation (as opposed to .h files)

A distinguishing philosophy

The Java™ Language Specification (1st edition, 1996)

Preface

Except for timing dependencies or other non-determinisms and given sufficient time and sufficient memory space, a Java program should compute the same result on all machines and in all implementations.

This philosophy includes floating-point

The Java™ Language Specification (1st edition, 1996)

4.2.3 Floating-Point Types and Values

The floating-point types are `float` and `double`, representing the single-precision 32-bit and double-precision 64-bit format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

This philosophy includes floating-point, cont.

The Java™ Language Specification (1st edition, 1996)

4.2.4 Floating-Point Operations

Operators on floating-point numbers behave exactly as specified by IEEE 754. In particular, Java requires support of IEEE 754 *denormalized* floating-point numbers and *gradual underflow*… Floating-point operations in Java do not "flush to zero" if the calculated result is a denormalized number.

[…]

Java requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision.

Compare with C/C++ undefined behavior

- In the Java platform:
  - Primitive types are all of known size and have defined overflow behavior
  - Cannot access uninitialed memory; array bounds checking required
  - …
- Undefined behavior on other platforms
  - *C and C++ Prioritize Performance over Correctness*, Russ Cox, 2023
  - boringcc, D. J. Bernstein, 2015
  - *What Every C Programmer Should Know About Undefined Behavior #1/3*, Chris Latter, 2011
  - …

A forward-looking statement from years ago

The JVM can be used for more than just the Java language.

The Java™ Virtual Machine Specification (1st edition, 1997)

Introduction

Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages. In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages.

Specifications and semantics

Java source code

javac

Other language source code

specification?

Scala

⋮

Kotlin

class files

JVM

IEEE 754 Compliant CPU

Detour: Background on the x87 FPU

Designed circa 1977

- With just 40,000 thousand transistors(!), you got:
  - Correctly rounded {+, –, ×, ÷, √}
  - *Plus* pretty-accurate transcendental functions
  - *But wait*, there's more, an indigenous 80-bit double extended format
    - Exp_{max} = 16383 (and Exp_{min} = –16382), 15 bits
    - p = 64 significand bits
  - Stack-based register set
- Predates IEEE 754 and has features not in IEEE 754 (e.g. projective infinity)

x87 control word

- The control word allows the processor to round to `float` or `double` *precision*, ***but*** using the registers' extended exponent range
- When rounding to double precision, values computed in registers equivalent to rounding to a format with
  - Exp_{max} = 16383 (and Exp_{min} = –16382), 15 bits
  - p = 53 significand bits
  - Intended to reduce incidence of intermediate over/underflow
- `double` has Exp_{max} = 1023 (and Exp_{min} = –1022), 11 bits
- Storing to memory can be used to round the exponent to `double`'s exponent range.

Is a store/(re)load after each *op* equivalent to strict `double`?

First approximation

- Pseudo assembly:

```
$op a b
store $XYZ
load  $XYZ
```

- To be fully equivalent, the computed result after *op*-store-reload needs to be equivalent for ***all*** inputs
  - For finite values, being equivalent implies the (*sign*, *significand*, *exponent*) triples must match
  - Infinity and NaN must result for the same inputs too

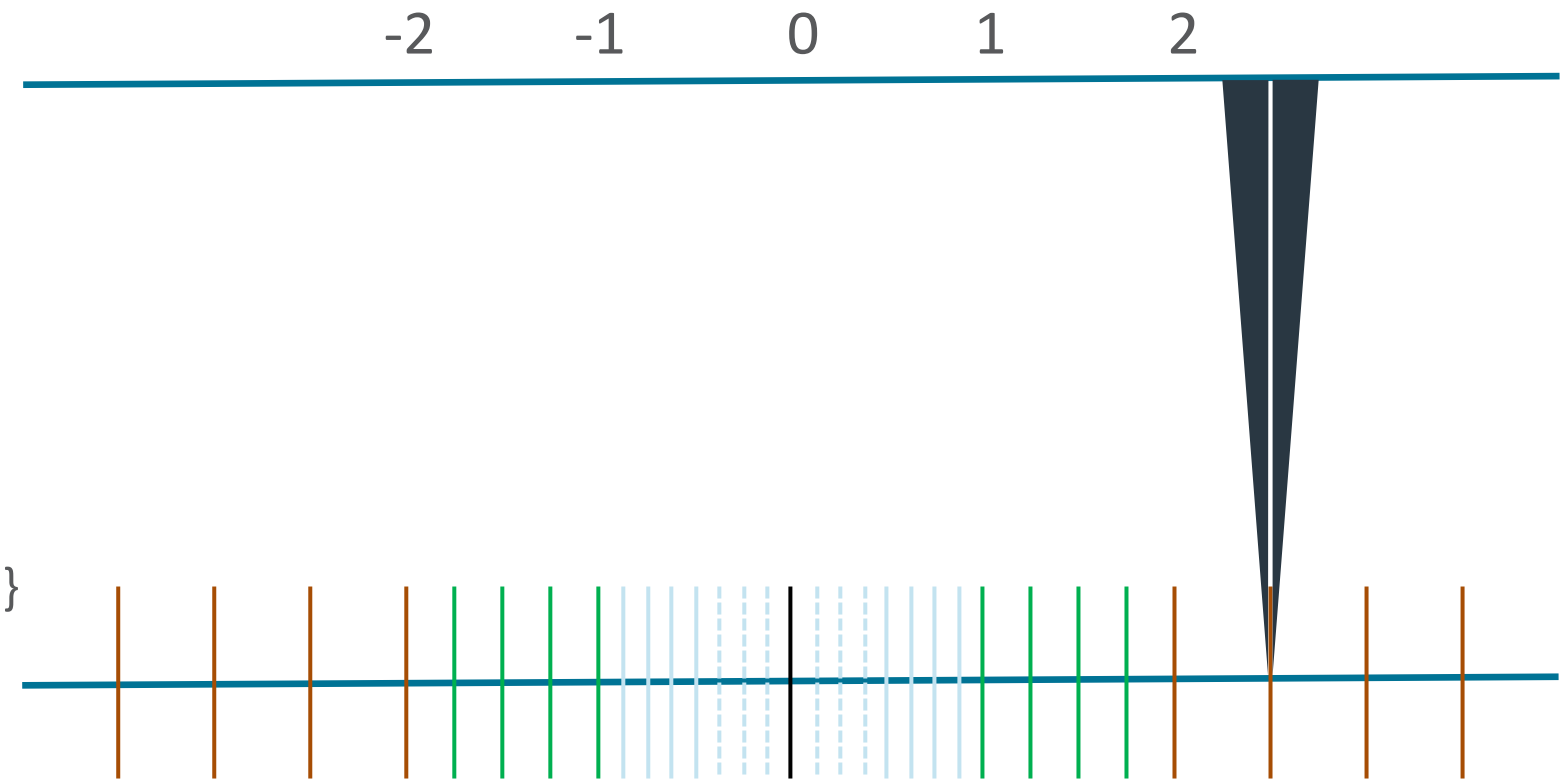
A problematic product example

- Consider the product of the two (carefully chosen) floating-point values:
  - `0x1.3b13b1p-1022` `(2.7385524106339 24E-308)`
  - `0xd.0000027p-4` `(0.8125000090803951)`
- Results:
  - Strict `double` product: `0x0.fffffffffffffp-1022` `(2.2250738585072 01E-308)`
  - Product w/ store-reload: `0x1.0p-1022` `(2.2250738585072 014E-308)`

What’s going on?

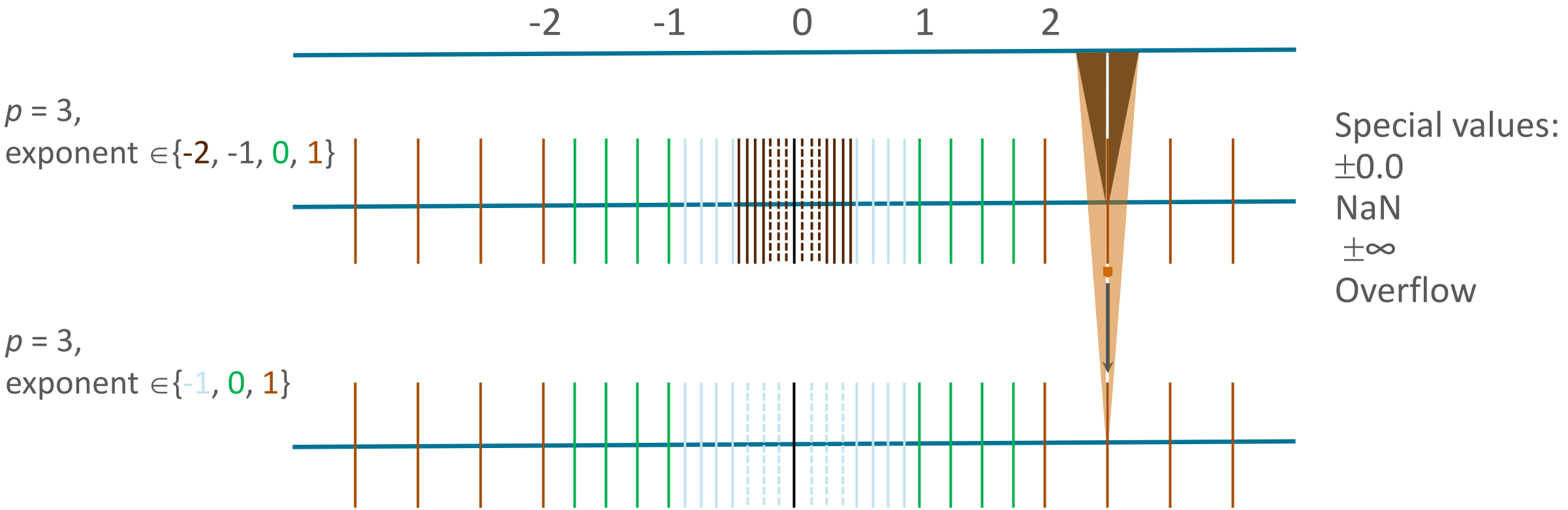
Look at an example with toy formats for guidance.

Case analysis: one rounding to final format



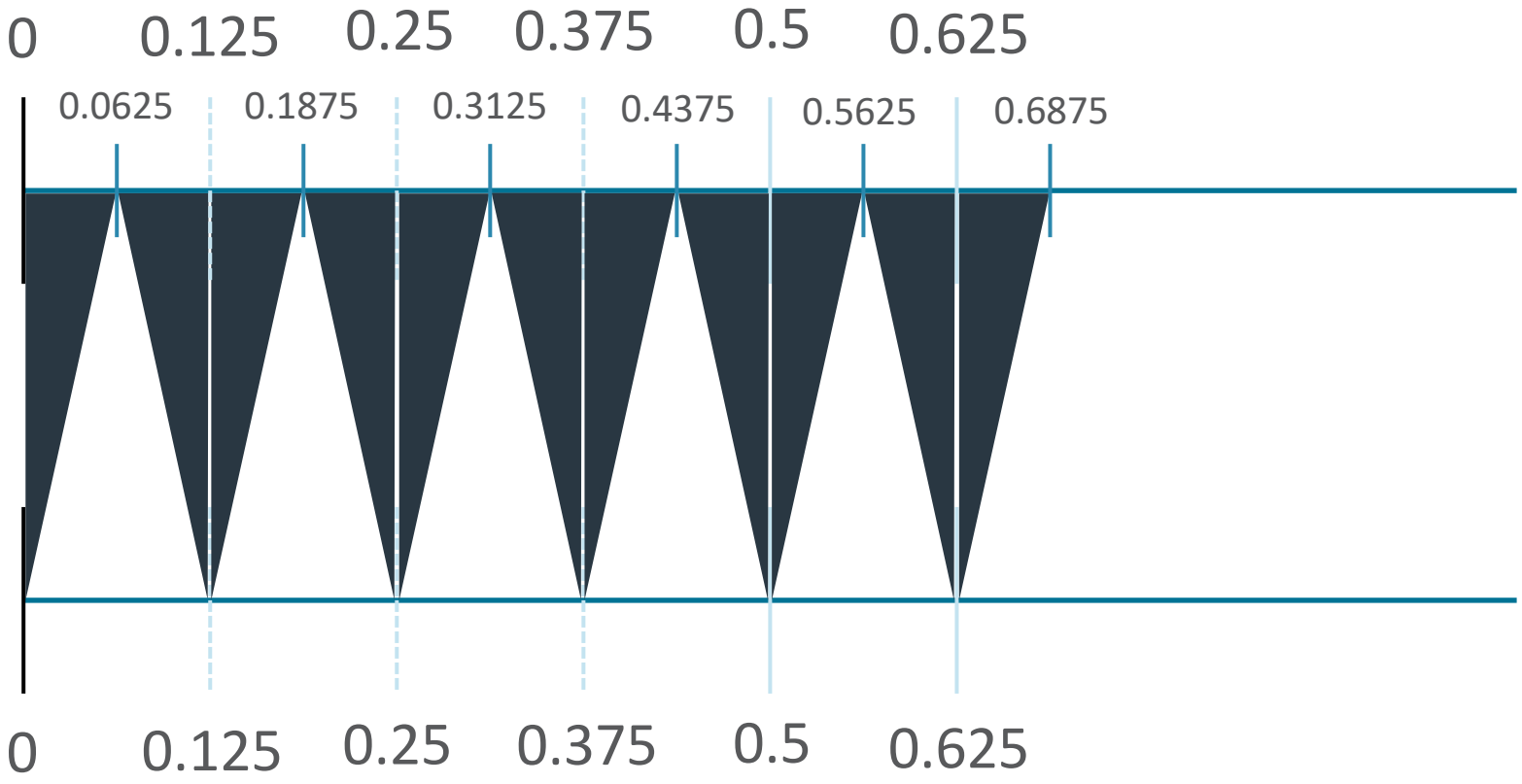
$p = 3$,
exponent $\in \{-1, 0, 1\}$

Case analysis: double rounding in the normal range



Rounding near minimum normal value of toy format

Representable values are evenly spaced



Toy format with p=3, Exp_{min} = –1

$0.0 = 0.00_2 \cdot 2^{-1}$

$0.125 = 0.01_2 \cdot 2^{-1}$

$0.25 = 0.10_2 \cdot 2^{-1}$

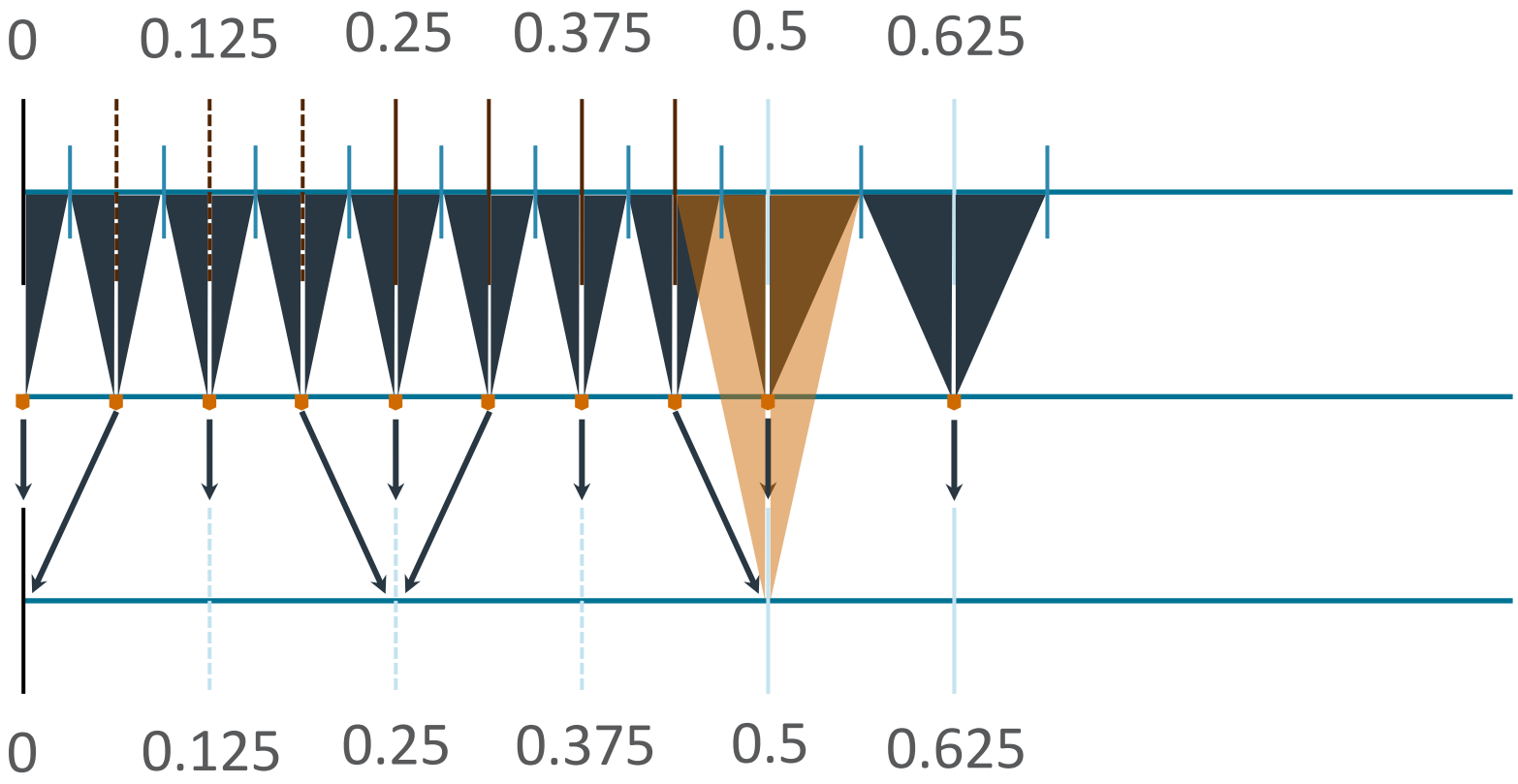
$0.375 = 0.11_2 \cdot 2^{-1}$

$0.5 = 1.00_2 \cdot 2^{-1}$

$0.625 = 1.01_2 \cdot 2^{-1}$

…

Are two roundings necessarily the same as one?



Toy format with p=3, Exp_{min} = –1

0.0 = $0.00_2 \cdot 2^{-1}$
0.125 = $0.01_2 \cdot 2^{-1}$
0.25 = $0.10_2 \cdot 2^{-1}$
0.375 = $0.11_2 \cdot 2^{-1}$
0.5 = $1.00_2 \cdot 2^{-1}$
0.625 = $1.01_2 \cdot 2^{-1}$
…

Toy format with p=3, Exp_{min} = –2

0.0625 = $0.01_2 \cdot 2^{-2}$
0.125 = $0.10_2 \cdot 2^{-2}$
0.1875 = $0.11_2 \cdot 2^{-2}$
0.25 = $1.00_2 \cdot 2^{-2}$
0.3125 = $1.01_2 \cdot 2^{-2}$
0.375 = $1.10_2 \cdot 2^{-2}$
0.4375 = $1.11_2 \cdot 2^{-2}$

Region near 0.5 with different results from double rounding: [0.40625, 0.4375).

Exact reproducibility problem

- Because the x87 did not support rounding to double *exponent* range when rounding to double *precision*, the *op* followed by store-reload code idiom has a double-rounding hazard for a tiny portion of the subnormal range.
- This is (literally) a small numerical difference, *but* it is large enough to run afoul of the explicit requirements of Java's language and virtual machine specifications.

For more, much more, information…

Shameless plug

Forward to the Past: The Case for Uniformly Strict Floating-Point Arithmetic on the JVM, JVM Languages Summit 2017, (slides), (video)

Abbreviated & abridged history of Java FP Proposals

- Java 1.0, May 1995; all strict all the time
- *The Evolution of Numerical Computing in Java*, James Gosling, circa 1997
- "*Proposal for Extension of Java ™ Floating Point Semantics, Revision 1, May 1998*" (PEJFPS) , IIRC document made public in August 1998
  - `strictfp` and `widefp`, `widefp` allows broad usage of wider *precision*, guts predictability of `widefp` expressions, which would have been the default
  - Press release courtesy archive.org

Concerning aspects of PEJFPS

- Within an FP-wide expression, format conversion allows an implementation, at its option, to perform any of the following operations on a value:
  - promote float to float extended and round float extended to float (note that double is a valid float extended format)
  - promote double to double extended and round double extended to double
- Not required to support subnormals in widefp methods

Reaction to PEJFPS

- After feedback from myself and others on `numeric-interest`, including the Java Grande document *Improving Java for Numerical Computation*, (October 1998) the PEJFPS proposal was replaced with the Java 1.2 `strictfp` and craftily redefined default floating-point semantics.
- Summary of Java 1.2 floating-point changes:
  - `strictfp` code has the original strict semantics
  - Default floating-point allows extended exponents (***not*** precision) for values on the JVM stack

Java 1.2 floating-point change in more detail

- Floating-point semantics were loosened the JLS and JVMS specifications to, in effect, allow the *op*-store-reload approximation to be used for expression evaluation.
  - By default, JVMs allowed to use extended exponent range for expression evaluation
  - Original strict semantics requested by `strictfp` Java language modifier and `ACC_STRICT` bit in the JVM.
  - Strict semantics implemented in x87 using a {scale ↓, *op,* scale ↑, store, reload} idiom developed by Roger Golliver of Intel.
  - Contributions from Java Grande effort, indirect influence of *JAVAHURT* presentation/document.
- *Predictability* preserved for floating-point expressions

A pragmatic compromise for the time

- Allowed reasonable performance by default;
  allowed strict reproducibility when needed
- *But*,
  - Complicated and brittle specifications
  - Some JVM implementations had optimization restrictions on inlining of methods with different `ACC_STRICT`/non- `ACC_STRICT` settings
  - Hard to use `strictfp` effectively in source

JEP 306: Restore Always-Strict Floating-Point Semantics

https://openjdk.org/jeps/306/

- "Make floating-point operations consistently strict, rather than have both strict floating-point semantics (`strictfp`) and subtly different default floating-point semantics. This will restore the original floating-point semantics to the language and VM, matching the semantics before the introduction of strict and default floating-point modes in Java SE 1.2."
- Delivered in JDK 17
  - Peculiarities of the original x87 architecture no longer a concern
  - Update to JLS, JVMS, HotSpot, `javac`, and core libraries
  - Facilitated final port of C FDLIBM to Java.
  - ACC_STRICT class file bit position available for other purposes in future work

Java Numerics: Possible Futures

Improving Java for Numerical Computation

Numerics Working Group of the Java Grande Forum, October 1998

	Issue	Requirement
1	*Complex arithmetic*	Complex numbers are essential in the analysis and solution of mathematical problems in many areas of science and engineering. Thus, it is essential that the use of complex numbers be as convenient and efficient as the use of float and double numbers.
2	*Lightweight classes*	Implementation of alternative arithmetic systems, such as complex, interval, and multiple precision requires the support of new objects with value semantics. Compilers should be able to inline methods that operate on such objects and avoid the overheads of additional dereferencing. In particular, lightweight classes are critical for the implementation of complex arithmetic as described in issue 1.
3	*Operator overloading*	Usable implementation of complex arithmetic, as well as other alternative arithmetics such as interval and multiprecision, requires that code be as readable as those based only on float and double.
4	*Use of floating-point hardware*	The high efficiency necessary for large-scale numerical applications requires aggressive exploitation of the unique facilities of floating-point hardware. At the other extreme, some computations require very high predictability, even at the expense of performance. The majority of users lie somewhere in between: they want reasonably fast floating-point performance but do not want to be surprised when computed results unpredictably misbehave. Each of these constituencies must be addressed.
5	*Multidimensional arrays*	Multidimensional arrays are the most common data structure in scientific computing. Thus, operations on multidimensional arrays of elementary numerical types must be easily optimized. In addition, the layout of such arrays must be known to the algorithm developer in order to process array data in the most efficient way.

Since then…

- *Use of floating-point hardware* largely handled by `strictfp` changes in JDK 1.2 and later implementation changes in use SSE and successors.
- Many other issues raised will be at least partially handled by Project Valhalla:
  1. *Value Objects*, introducing class instances that lack identity
  2. *Flattened Heap Layouts for Value Objects*, supporting null-free flattened storage
  3. *Enhanced Primitive Boxing*, allowing primitives to be treated more like objects
  4. *Null-Restricted and Nullable Types*, providing language support for managing nulls
  5. *Parametric JVM*, preserving and optimizing generic class and method parameterizations at runtime

Other numerically-related works in progress

Besides Valhalla

- Vector API
- Project Panama – Foreign Function & Memory API (preview) – better/faster JNI (Java Native Interface) replacement and much more
- Coming soon… *Project Babylon*
  Babylon's primary goal will be to extend the reach of Java to foreign programming models such as SQL, differentiable programming, machine learning models, and GPUs. Babylon will achieve this with an enhancement to reflective programming in Java, called code reflection.
- On-going incremental improvements to the JVM

Summary

- There and back again journey on strict semantics
- Ambient performance improvements over time from on-going work
- Multiple long-term projects to improve the Java platform
- Platform stewardship with attention to even the smallest details

Questions?

Thank you!

Slides will be linked to off of
https://github.com/jddarcy/SpeakingArchive