ORACLE

Exploring speedup opportunities in the GraalVM compiler

CGO Graal Workshop, Feb 27th 2021

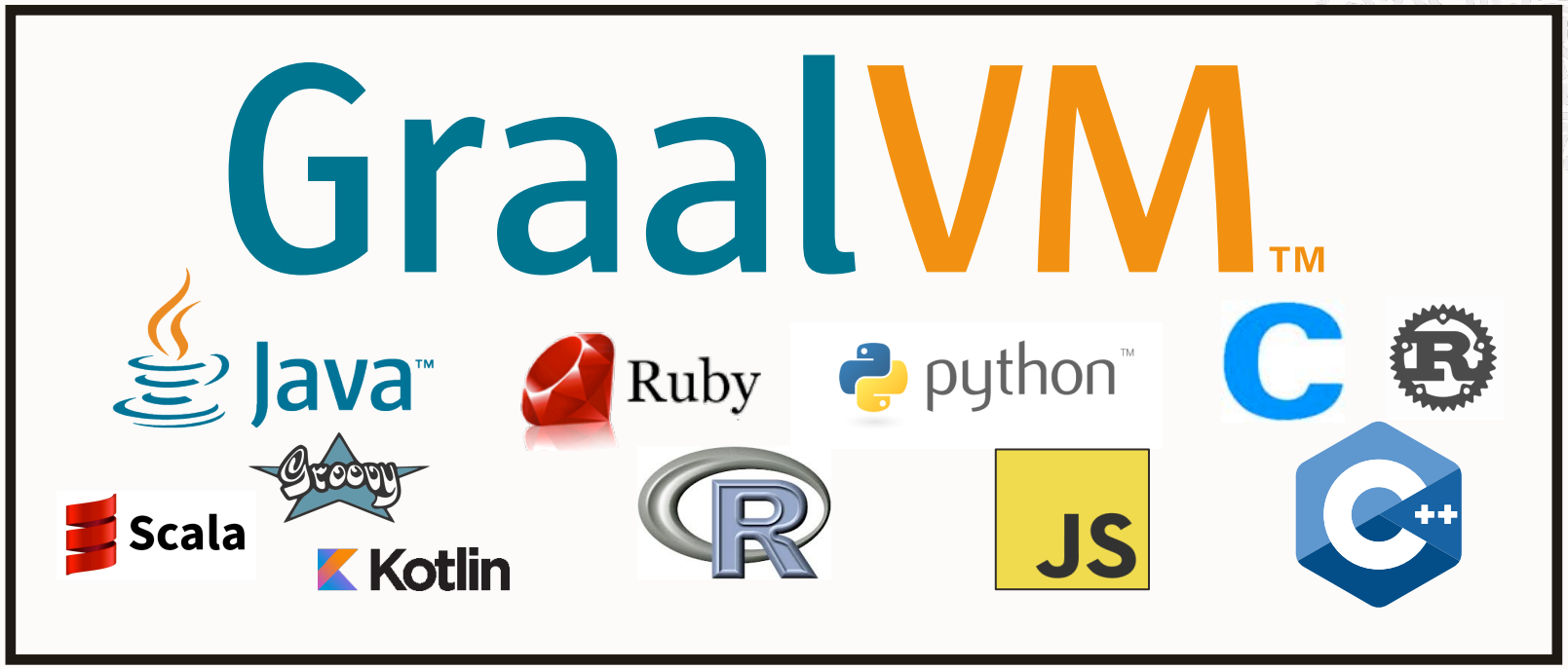
François Farquet
Principal Performance Engineer
GraalVM compiler team
Oracle Labs, Switzerland

@FFarquet

Safe Harbor Statement

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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image technology (including Substrate VM) is Early Adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification.

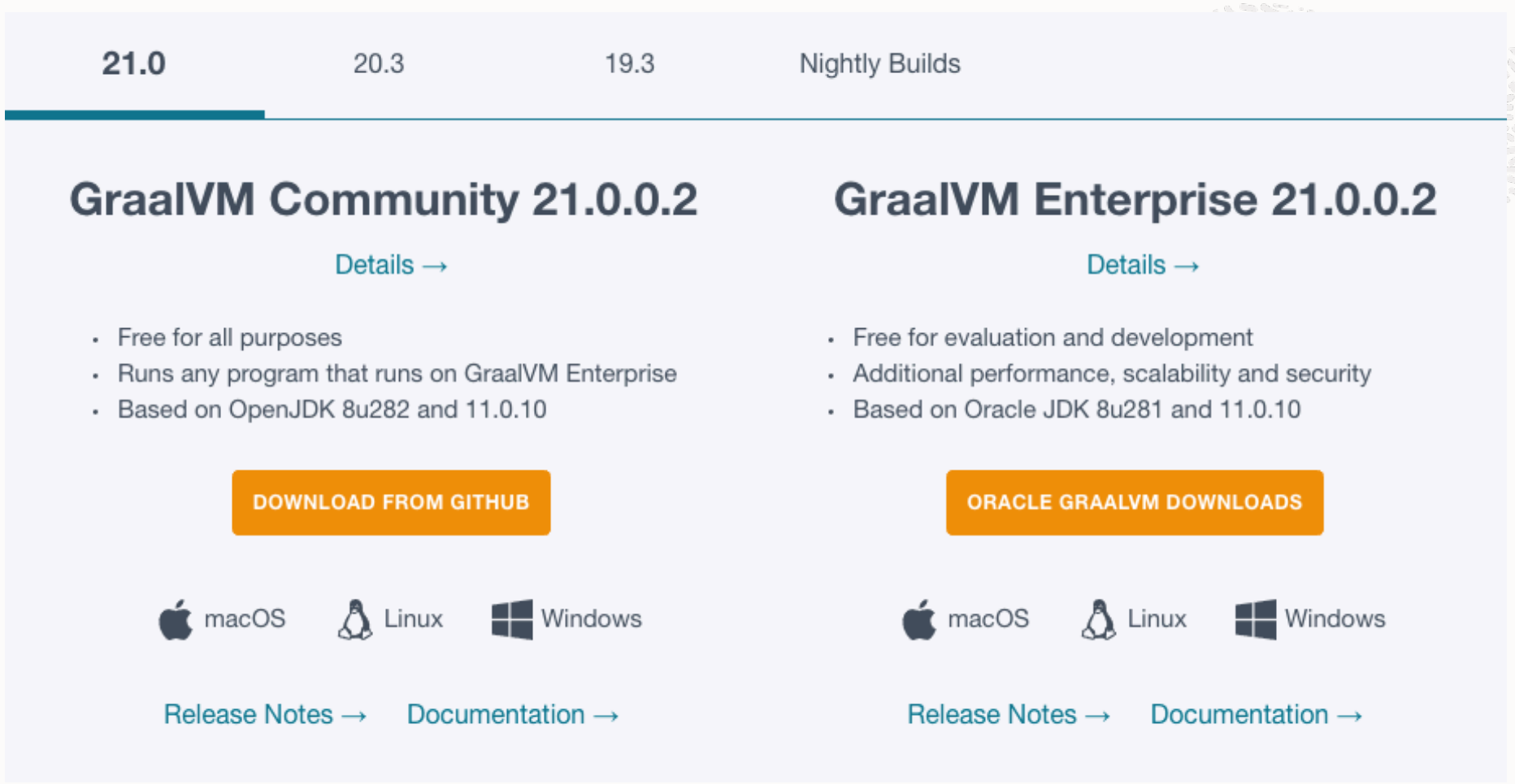


OpenJDK™

node.js

ORACLE DATABASE

standalone



https://www.graalvm.org/downloads/

Improve JIT compiler performance

Main metrics to optimize for:

- **Peak performance**
  - Throughput and/or latency at steady state
- **Warmup time**
  - time to reach peak performance

Other metrics to watch:

- Compilation time
- Code size installed
- Memory footprint

Improve JIT compiler performance

How?

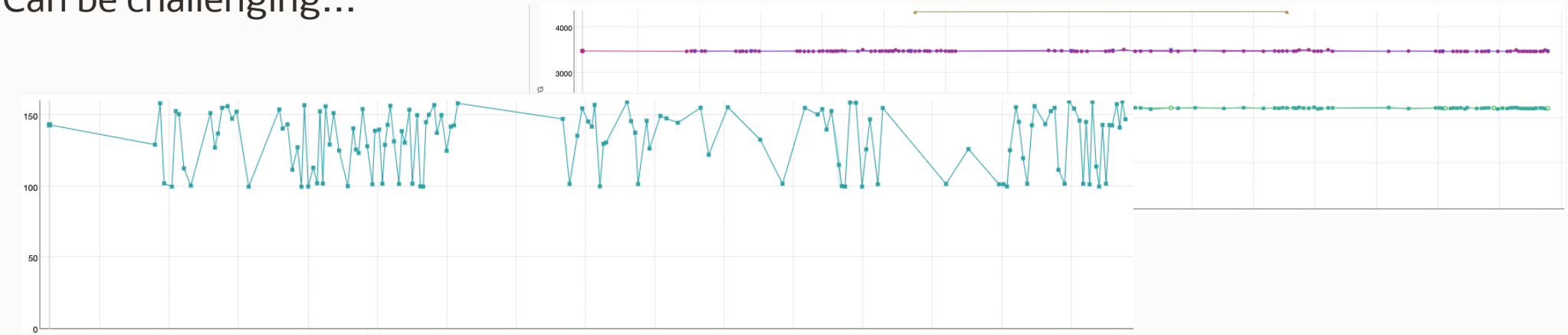
1. **Compiler R&D**
   - apply novel techniques
   - Explore graph patterns
   - add new intrinsics
   - Investigate important patterns reported by customers/community
   - Optimize new technology: Scala, Java Streams, popular frameworks
   - …

2. **Day to day performance tracking**

3. **Chasing opportunities within the GraalVM compiler**

Day-to-day performance tracking

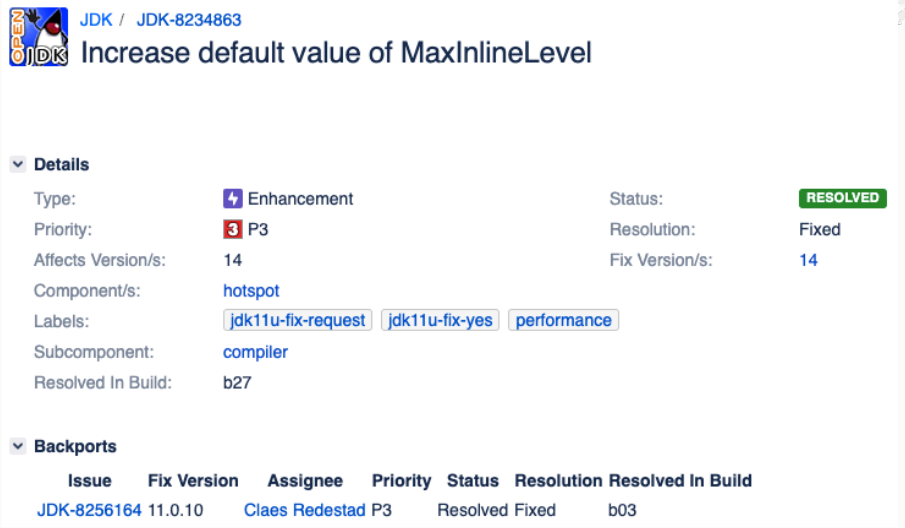
- **Regression tracking**
  - Make sure that today's performance is better or equal than yesterday's
  - Can be challenging…



  - The reason can be:
    - the JVM (often GC or compiler)
    - The benchmark
    - The infrastructure

Day-to-day performance tracking

- **Baseline comparison against HotSpot**
  - It's a moving target !
    - JDK libraries change
    - HotSpot improves



If HotSpot outperforms GraalVM in a benchmark, this is a good news!

It usually means there is a low-hanging fruit to optimize the GraalVM compiler further.

Chasing opportunities within the GraalVM compiler

Play with command line flags:

```
java -XX:+PrintFlagsFinal -XX:+JVMCIPrintProperties -version
```

Flags	OpenJDK 11.0.9	GraalVM CE 20.3	GraalVM EE 20.3
-XX:*	668	668	668
-Dgraal.*	-	**248**	**554**

Graal flags categories

1. **Debugging and tracing flags**: enable tracing, dumping, printing, method filters, etc

2. **Debug compiler phases by forcing a decision**, skipping the heuristic

3. **Enable/Disable an optimization** or a compiler phase

4. **Tweak heuristics** Modifying the behavior of the compiler

```
…
graal.OptDevirtualizeInvokesOptimistically = true                    [Boolean]
graal.OptEarlyReadElimination = true                                 [Boolean]
graal.OptEliminateGuards = true                                      [Boolean]
graal.OptFloatingReads = true                                        [Boolean]
graal.OptImplicitNullChecks = true                                   [Boolean]
graal.OptReadElimination = true                                      [Boolean]
graal.OptScheduleOutOfLoops = true                                   [Boolean]
graal.PartialEscapeAnalysis = true                                   [Boolean]
graal.PartialUnroll = true                                           [Boolean]
…
```

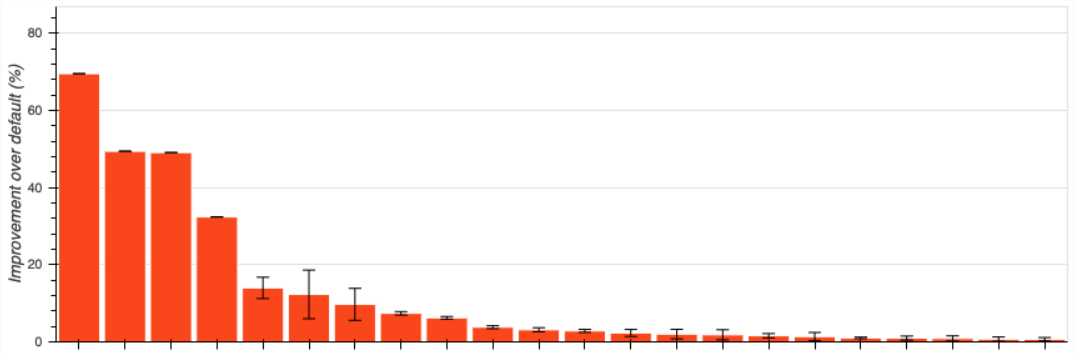
```
@Option(help = "", type = OptionType.Expert) public static final OptionKey<Integer> FullUnrollMaxNodes = new OptionKey<>(400);
@Option(help = "", type = OptionType.Expert) public static final OptionKey<Integer> FullUnrollConstantCompareBoost = new OptionKey<>(15);
@Option(help = "", type = OptionType.Expert) public static final OptionKey<Integer> FullUnrollMaxIterations = new OptionKey<>(600);
@Option(help = "", type = OptionType.Expert) public static final OptionKey<Integer> ExactFullUnrollMaxNodes = new OptionKey<>(800);
@Option(help = "", type = OptionType.Expert) public static final OptionKey<Integer> ExactPartialUnrollMaxNodes = new OptionKey<>(200);
```

https://github.com/oracle/graal/blob/master/compiler/src/org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/loop/DefaultLoopPolicies.java

Debug compiler phases by forcing a decision

- *-Dgraal.PeelALot=true* (default: false)
  - forces the peeling of all candidate loops
- Found several microbenchmarks where performance improved greatly when enabling this debug flag
- **Missed opportunity to hoist instanceofs/checkcasts out of loops**
- Fixed in:

  Improve `InstanceOfNode` anchoring.
  https://github.com/oracle/graal/commit/20c3421a0da0168876957f614f91dacd696252b3



Tweak heuristics

Computationally, it becomes more challenging if one wants to explore numeric fields.
To explore a set of values of a single option, it requires:

```
B*M*N*T minutes
```

- *B* = number of benchmarks to test
- *M* = number of values to test
- *N* = number of experiment repetitions to get reliable numbers
- *T* = average time to run a benchmark

For a single value to test (*M=1*) and to get a full picture, one may want:
B = 60 (for DaCapo, ScalaBench, Renaissance, SpecJVM2008)
N = 10
T = 5 minutes

```
3000 minutes = 50 machine hours
```

For a single value of a single option !

Tweak heuristics

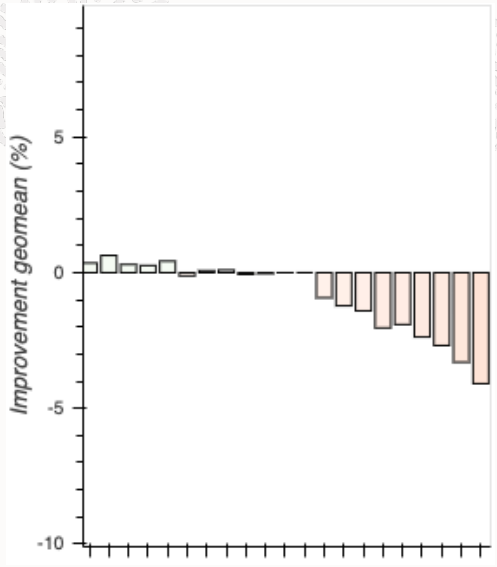
After weeks of benchmarking, we understood better how GraalVM Enterprise compiler reacts to its inliner option values.

- Led to better default values for some parameters
  - Improving some benchmarks from Renaissance, ScalaBench, SpecJVM2008
  - Peak performance improved **by 1% to 40%**
  - at no extra compilation time or warmup cost

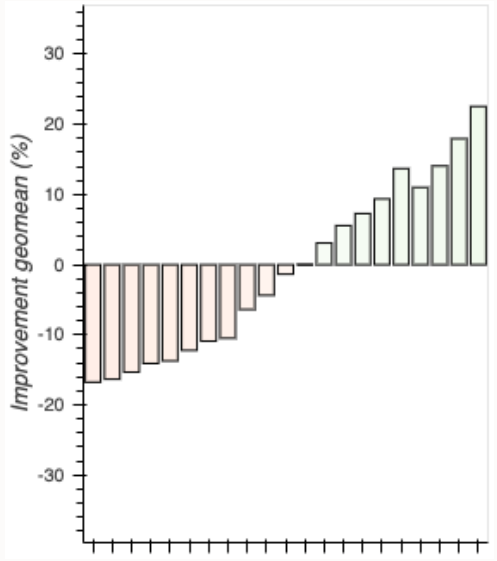
Tweak heuristics

However, there are other options that lead to great peak performance improvements at some compilation time cost.

Performance change

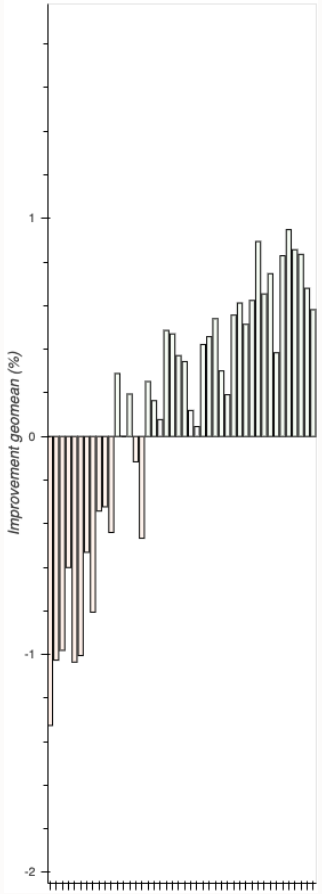


Compilation time

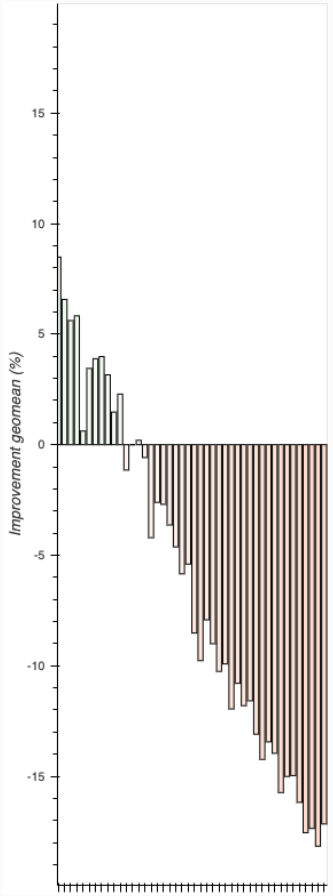


Tweak heuristics

Peak performance improvement



Compilation time impact

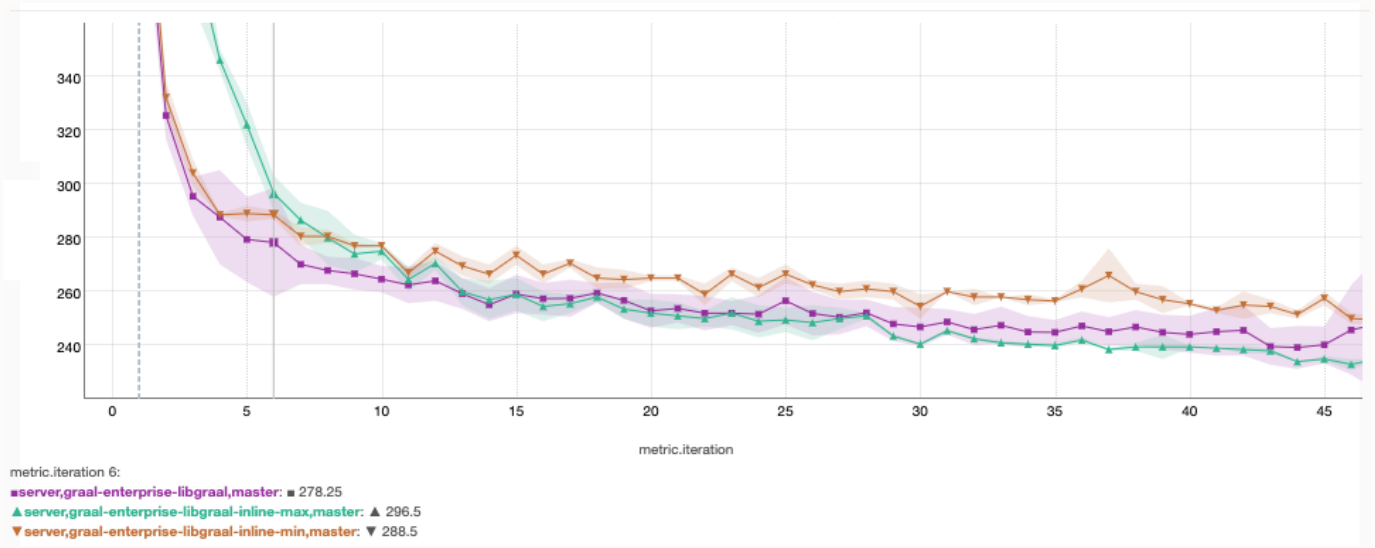


Tweak heuristics

Let's expose those trade-offs to the user!

Introducing a new GraalVM Enterprise flag **–Dgraal.TuneInlinerExploration=[-1,1]** (default: 0)

The closer the value is to -1, the less aggressive the inliner, the closer it is to 1, the more aggressive it is.



Thank you!

@FFarquet