In collaboration with

EPFL ORACLE

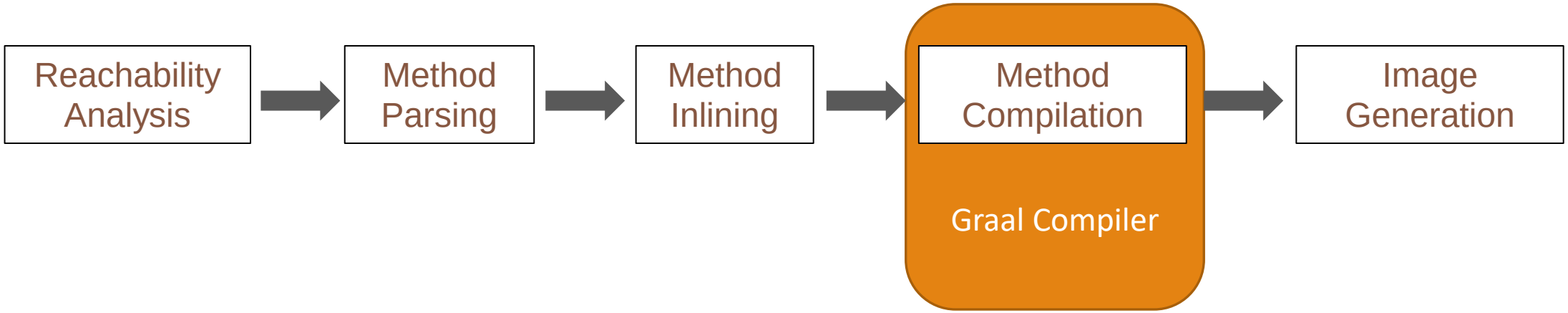
Quick build Mode

BUILD TIME IMPROVEMENTS IN NATIVE IMAGE

Presenter: Carlo Refice (crefice@gmail.com)

Intro – Native Image

Compile a whole Java application to a native executable ahead-of-time for increased performance.



Graal compiler

- Custom “Sea of nodes” IR
- Code transformation phase grouped in *Tiers*
- *Lowering* performs required transformations between tiers

Graal phase plan

High Tier

GraphBuilder
Inlining
DeadCodeElimination
DisableOverflownCountedLoops
ConvertDeoptimizeToGuard
LoopFullUnroll
LoopPeeling
LoopUnswitching
BoxNodeIdentity
PartialEscape
ReadElimination
BoxNodeOptimization

Mid Tier

LockElimination
FloatingRead
ConditionalElimination
LoopSafepointElimination
SpeculativeGuardMovement
GuardLowering
LoopFullUnroll
RemoveValueProxy
LoopSafepointInsertion
OptimizeDiv
FrameStateAssignment
LoopPartialUnroll
Reassociation
DeoptimizationGrouping
Canonicalizer
WriteBarrierAddition

Low Tier

ExpandLogic
FixReads
Canonicalizer
UseTrappingNullChecks
DeadCodeElimination
PropagateDeoptimizeProbability
Schedule

Graal phase plan

High Tier

GraphBuilder
Inlining
DeadCodeElimination
DisableOverflownCountedLoops
ConvertDeoptimizeToGuard
LoopFullUnroll
LoopPeeling
LoopUnswitching
BoxNodeIdentity
PartialEscape
ReadElimination
BoxNodeOptimization

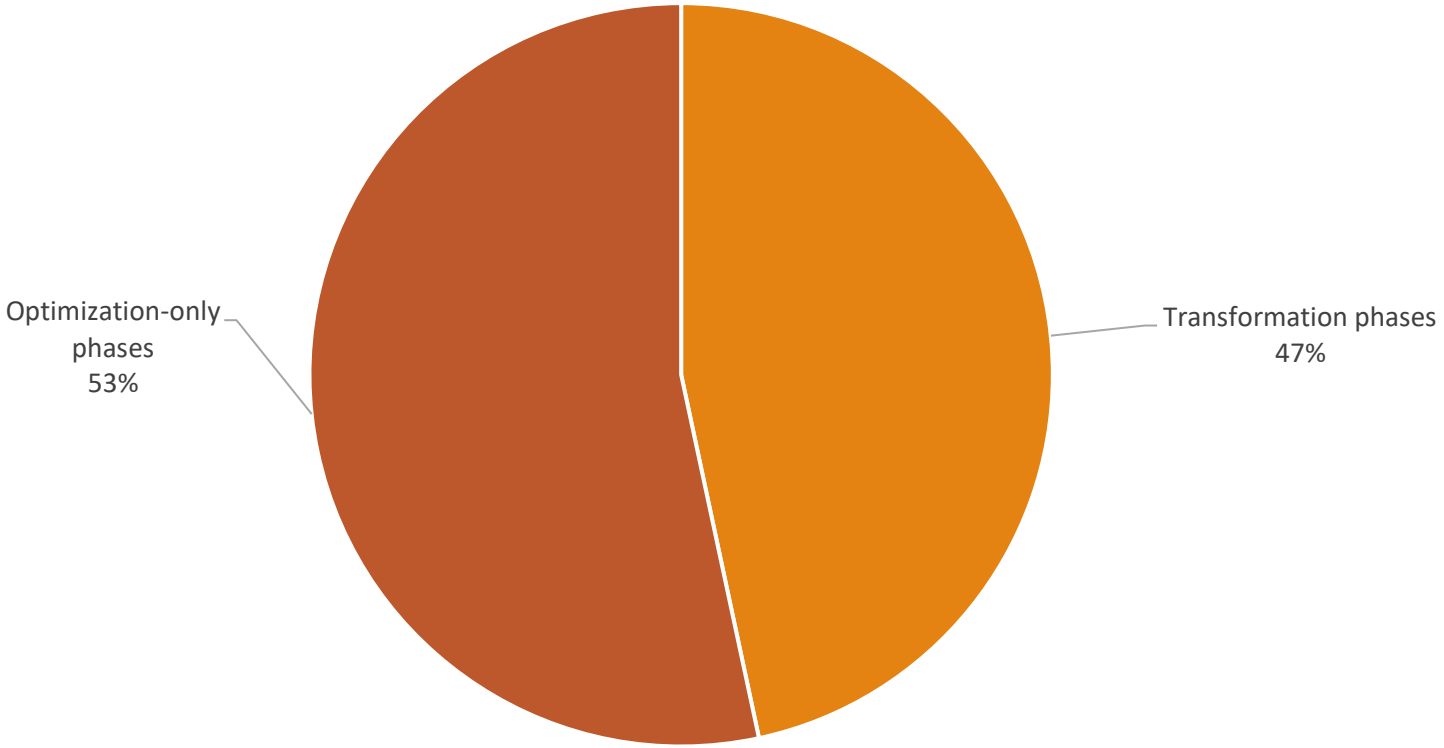
Mid Tier

LockElimination
FloatingRead
ConditionalElimination
LoopSafepointElimination
SpeculativeGuardMovement
GuardLowering
LoopFullUnroll
RemoveValueProxy
LoopSafepointInsertion
OptimizeDiv
FrameStateAssignment
LoopPartialUnroll
Reassociation
DeoptimizationGrouping
Canonicalizer
WriteBarrierAddition

Low Tier

ExpandLogic
FixReads
Canonicalizer
UseTrappingNullChecks
DeadCodeElimination
PropagateDeoptimizeProbability
Schedule

Graal phase timings



Idea – Economy phase plan

Streamlined phase plan: fast startup, slow runtime

Meant as:

- First level of multi-tier compilation in Truffle
- A replacement for the C1 compiler in Java tiered compilation

Optimization levels

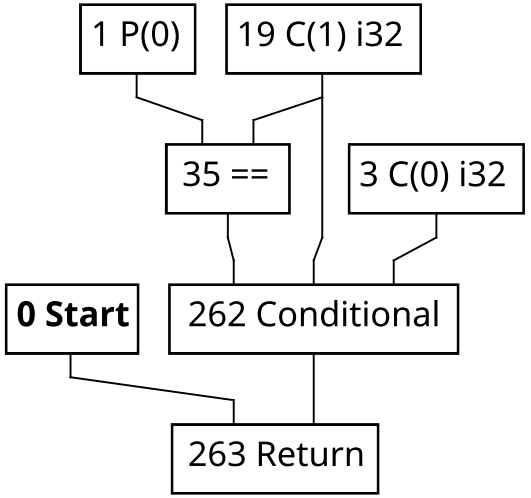
```
public static boolean test(int arg) {
    int x = arg;

    for (int i = 0; i < 10; i++) {
        int y = m();
        if (x == 1) {
            return true;
        }
        x = y;
    }
    return false;
}
```

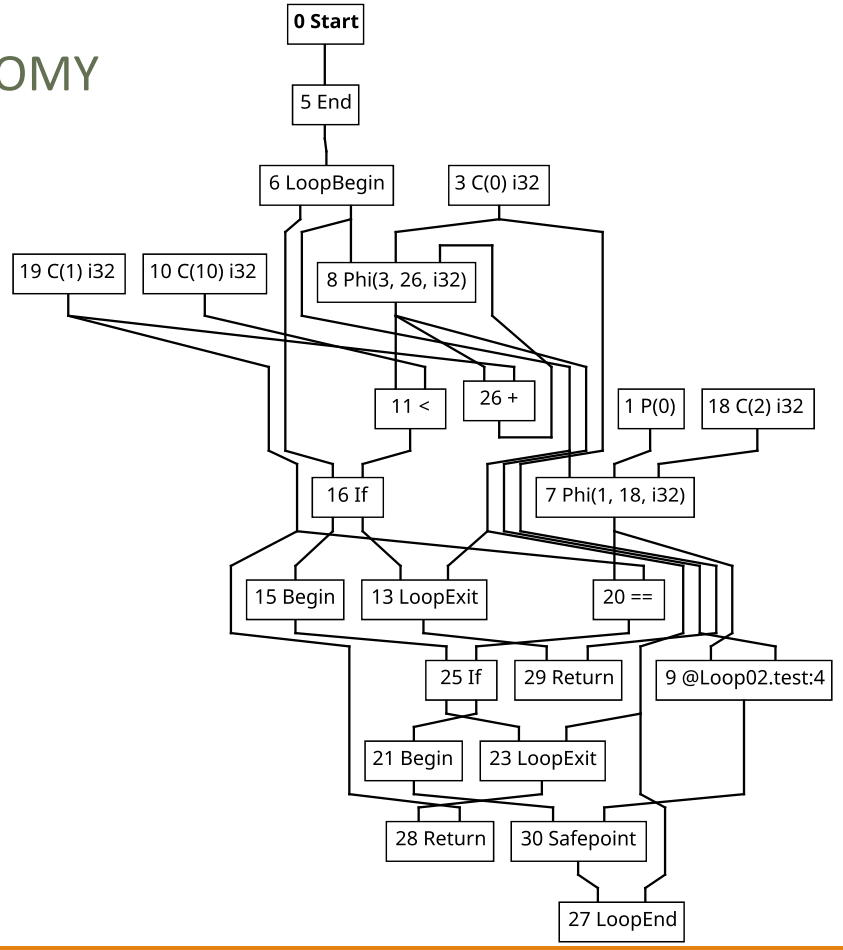
```
private static int m() {
    return 2;
}
```

Optimization levels

DEFAULT

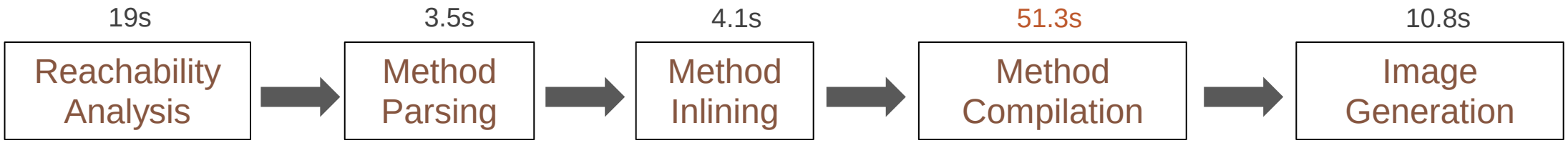


ECONOMY



Native Image

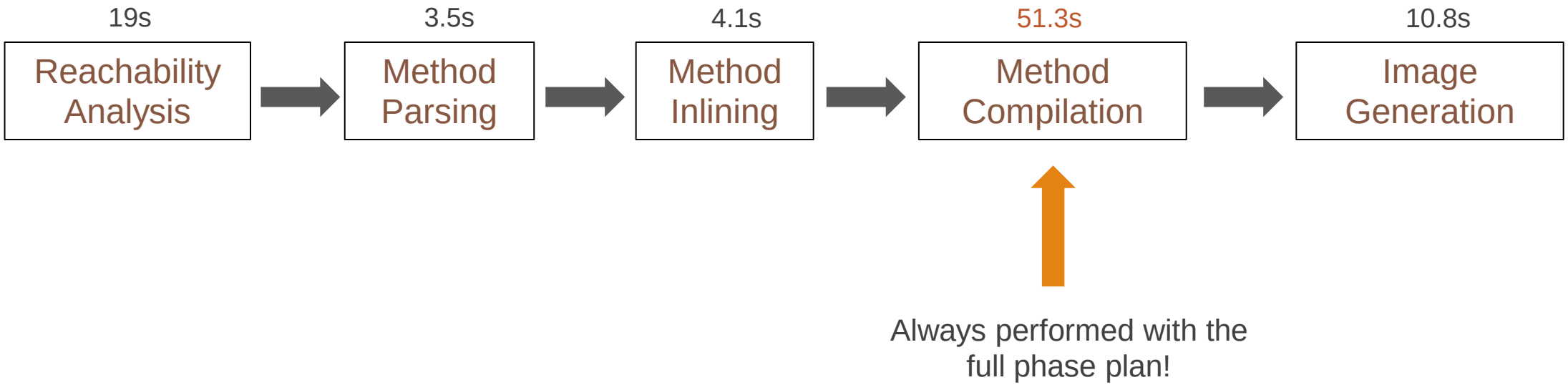
Compile a whole Java application to a native executable ahead-of-time for increased performance.



(GraalVM EE)

Native Image

Compile a whole Java application to a native executable ahead-of-time for increased performance.



(GraalVM EE)

Idea – "Quick build Mode"

For faster build times, image could be compiled with Economy mode instead.

User flag –Ob (Optimize for build time):

```
$ native-image -Ob HelloWorld.java helloworld
```

Intended as a Development-focused mode for fast iteration time

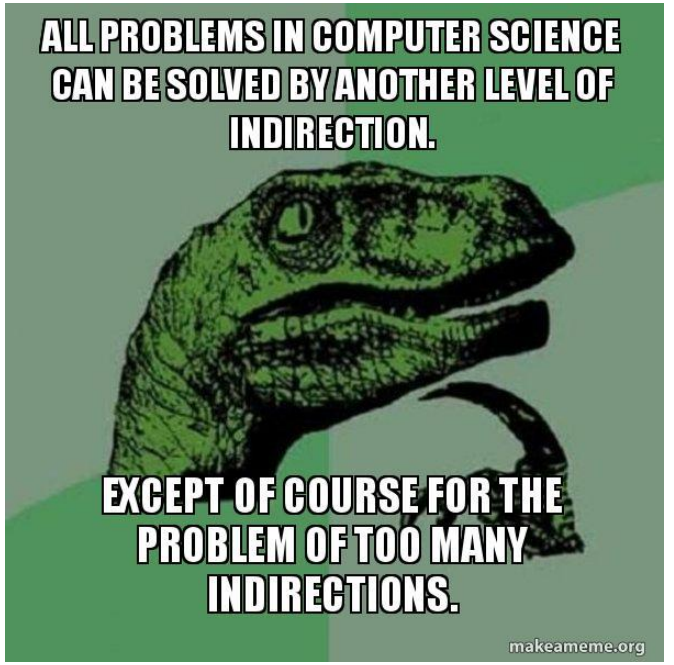
Hosted vs Runtime compilation

A lot of the configuration of the image build process happens through singletons that are shared between Hosted (build) and Runtime.

This is especially a problem when building enterprise Libgraal: we still want a compiler built without enterprise features to keep its enterprise features at runtime!

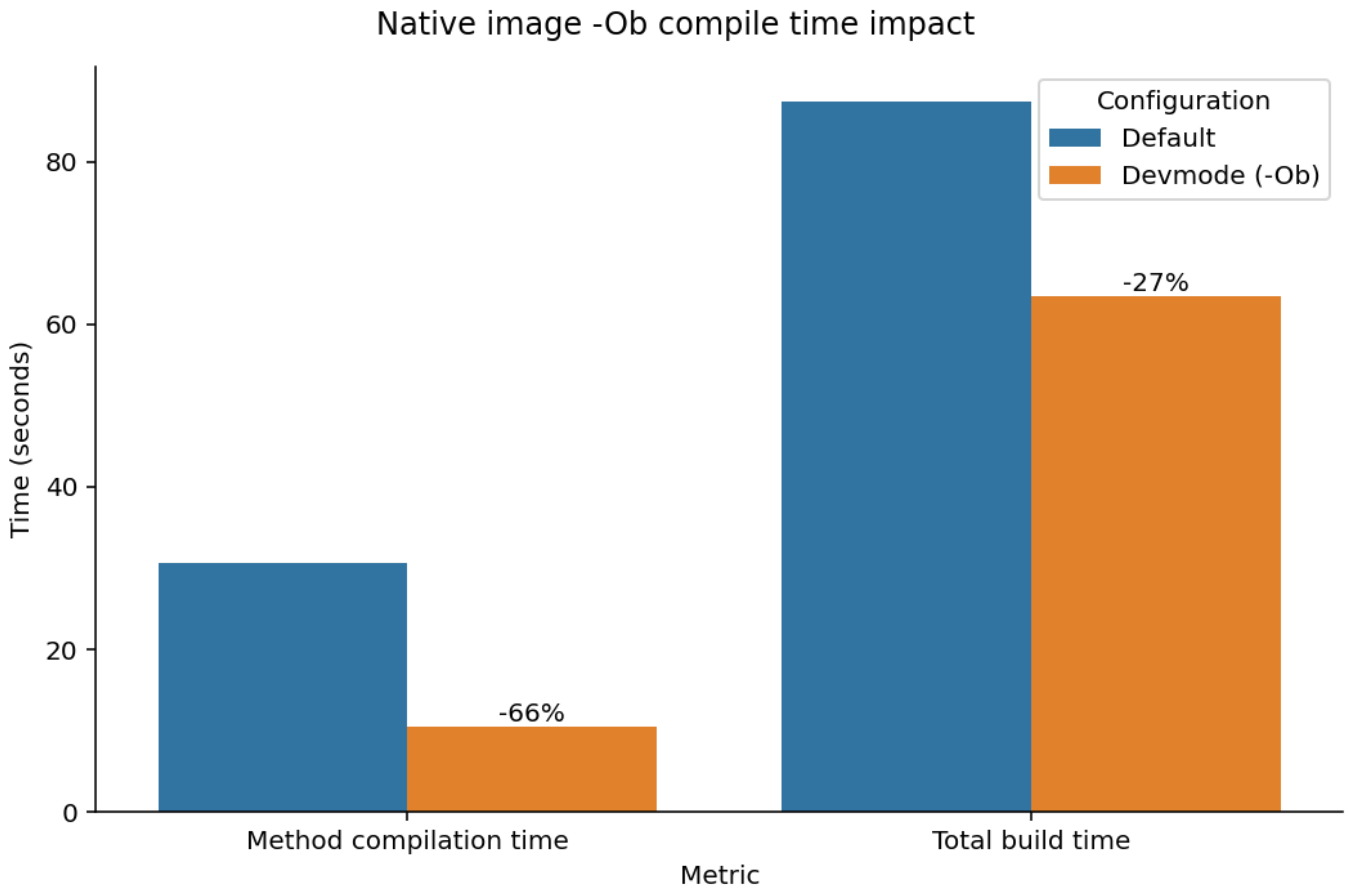
Thankfully, adding one layer of indirection solved the problem.

https://medium.com/graalvm/libgraal-graalvm-compiler-as-a-precompiled-graalvm-native-image-26e354bee5c



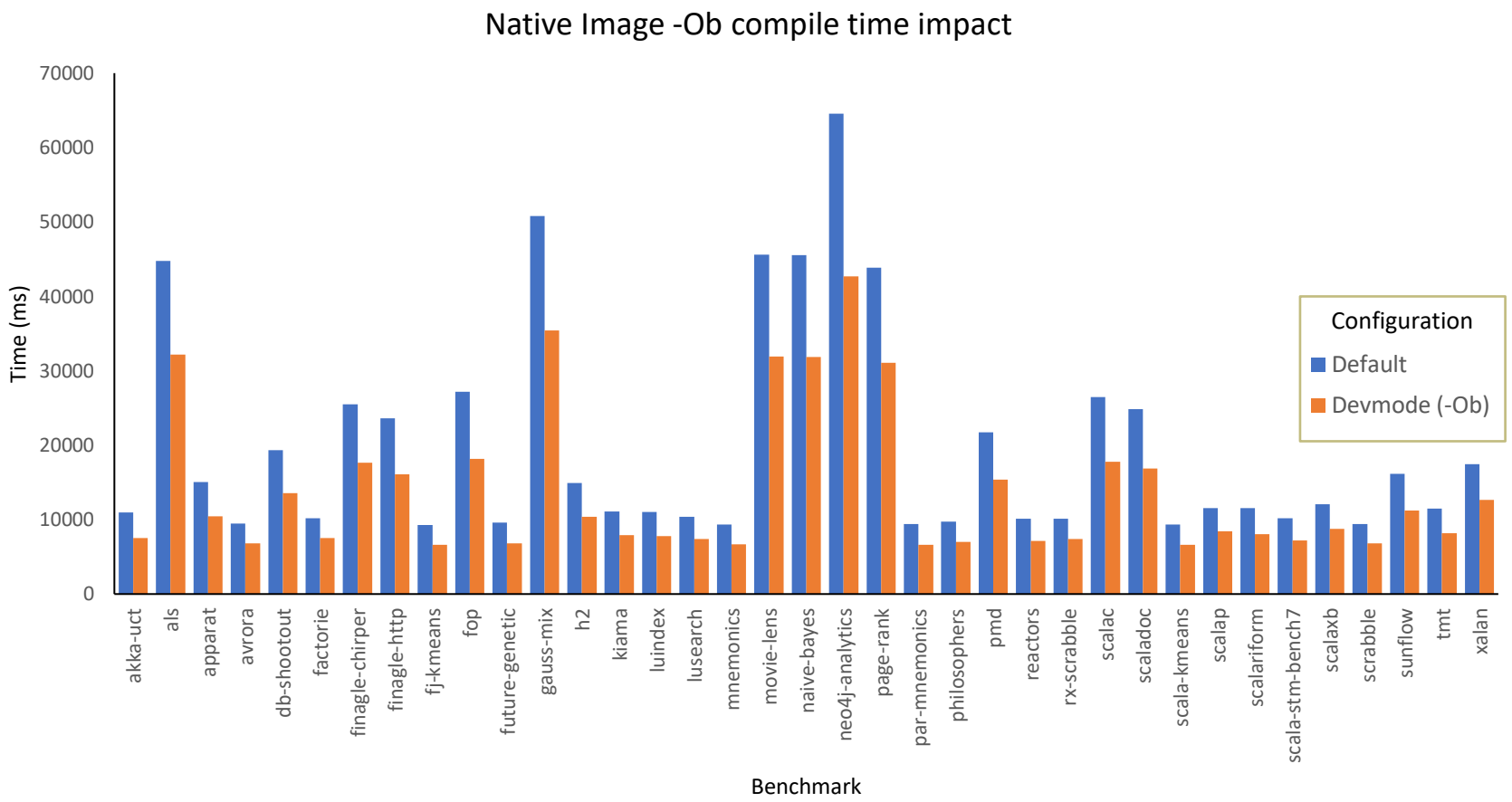
Benchmark Results

Compile Time



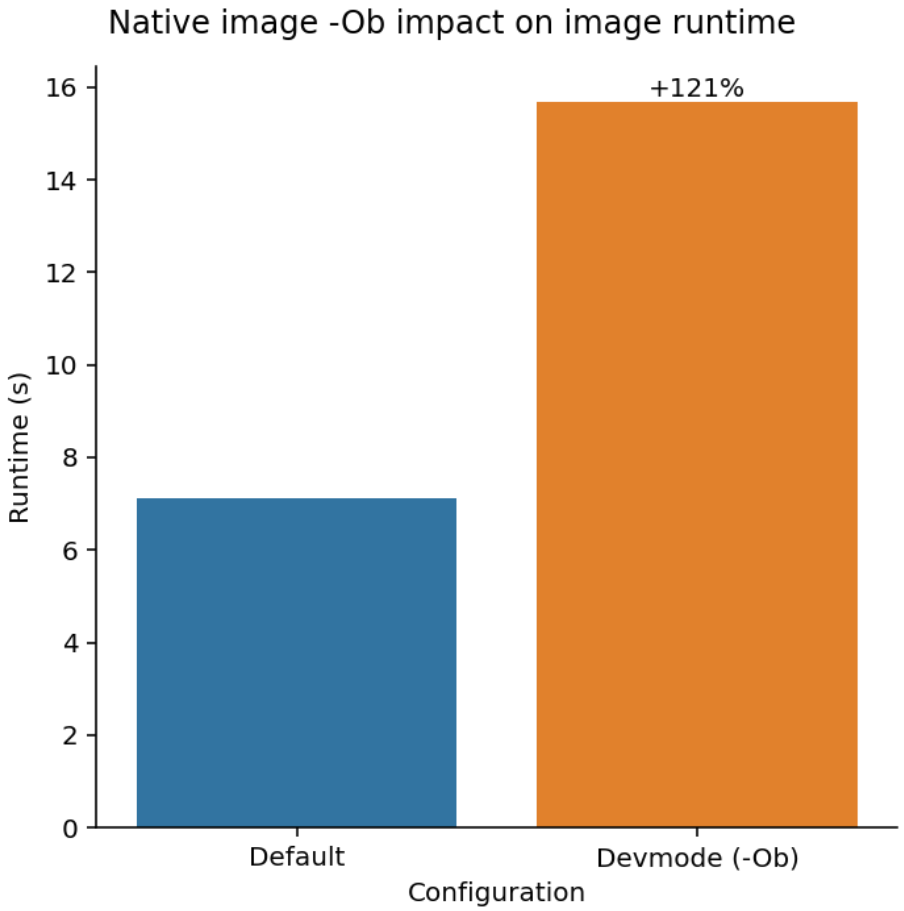
Benchmark Results

Compile Time



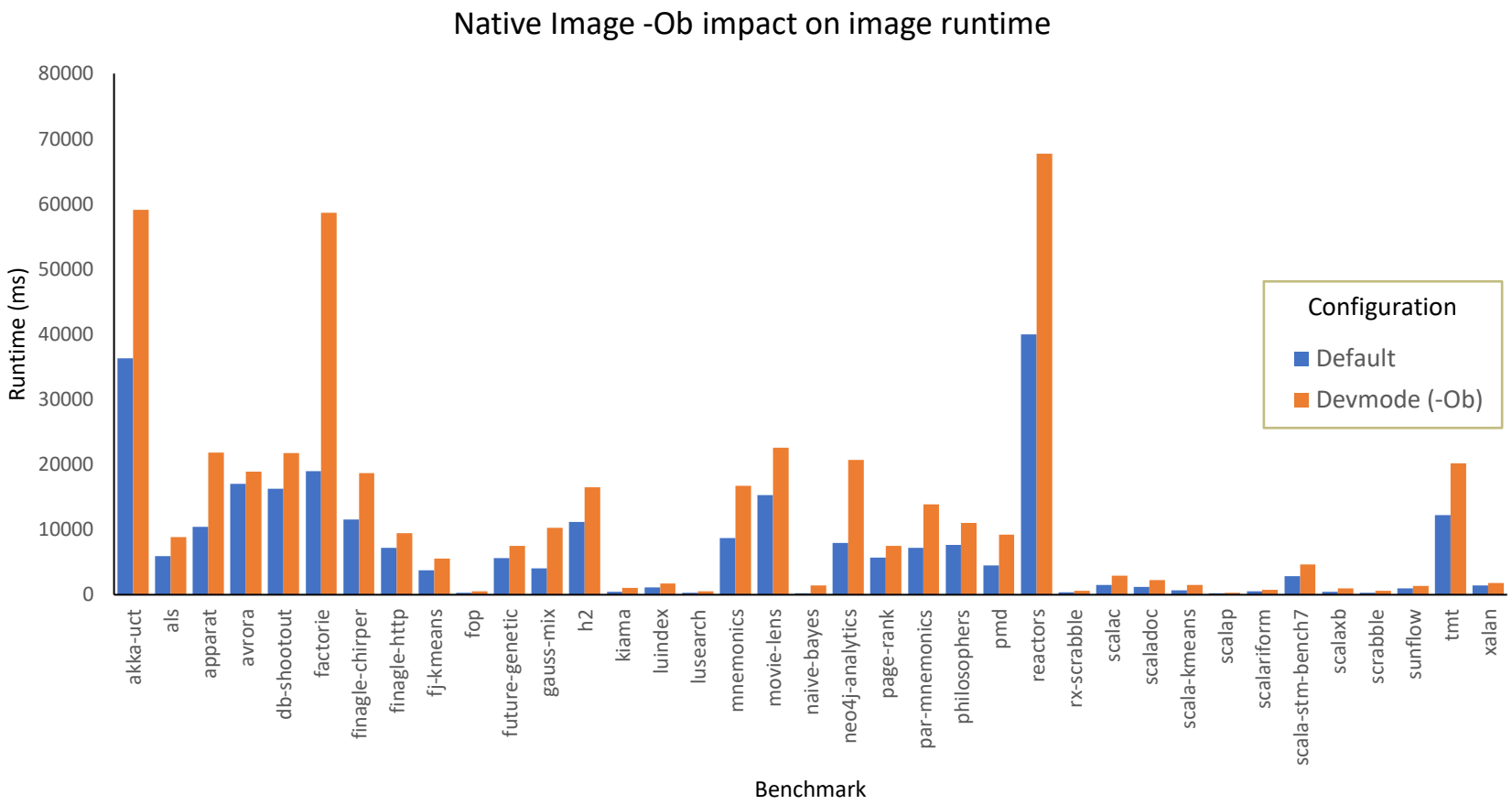
Benchmark Results

Runtime performance



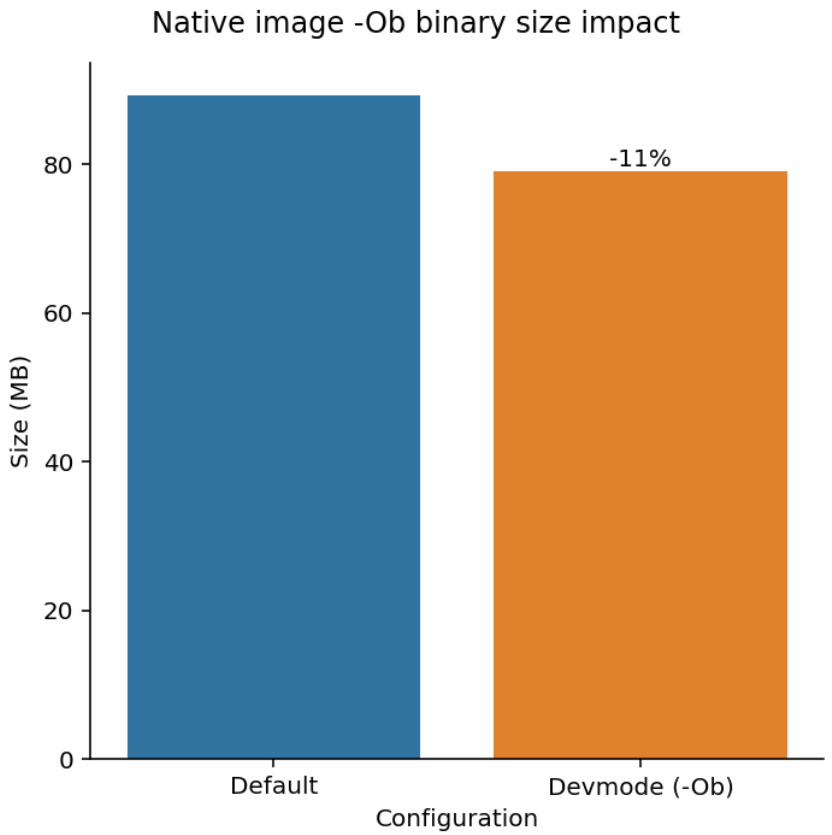
Benchmark Results

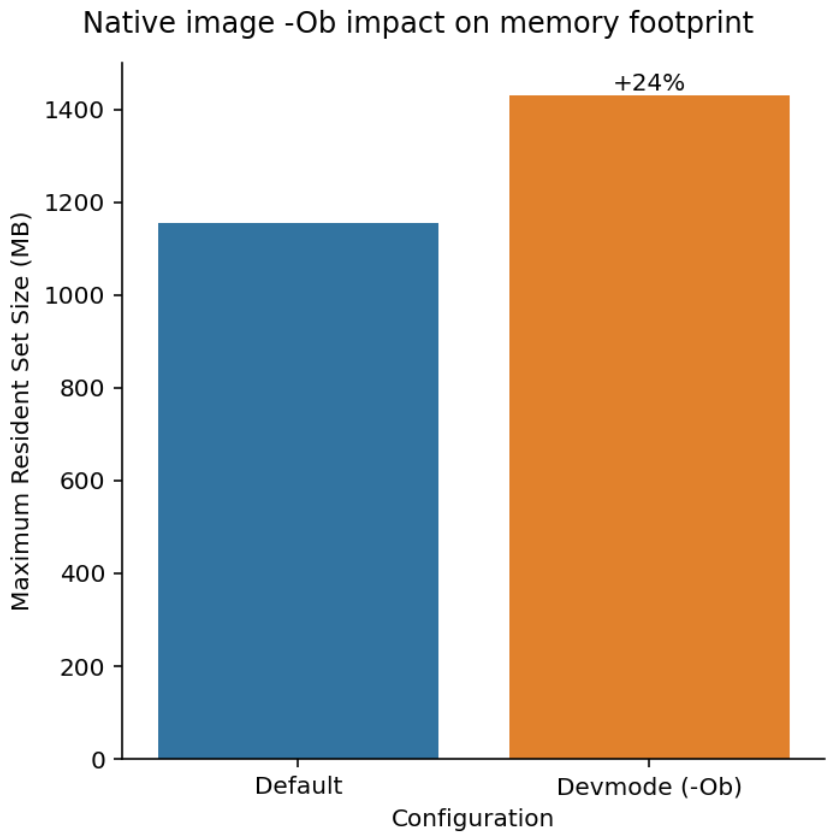
Runtime performance



Benchmark Results

Memory and image size





Future Work

- Further build time improvements, e.g. in analysis stage
- Tweak performance to reduce bottlenecks in certain benchmarks
- Enable by default rather than on-demand.

Thank you for listening!