
ORACLE®

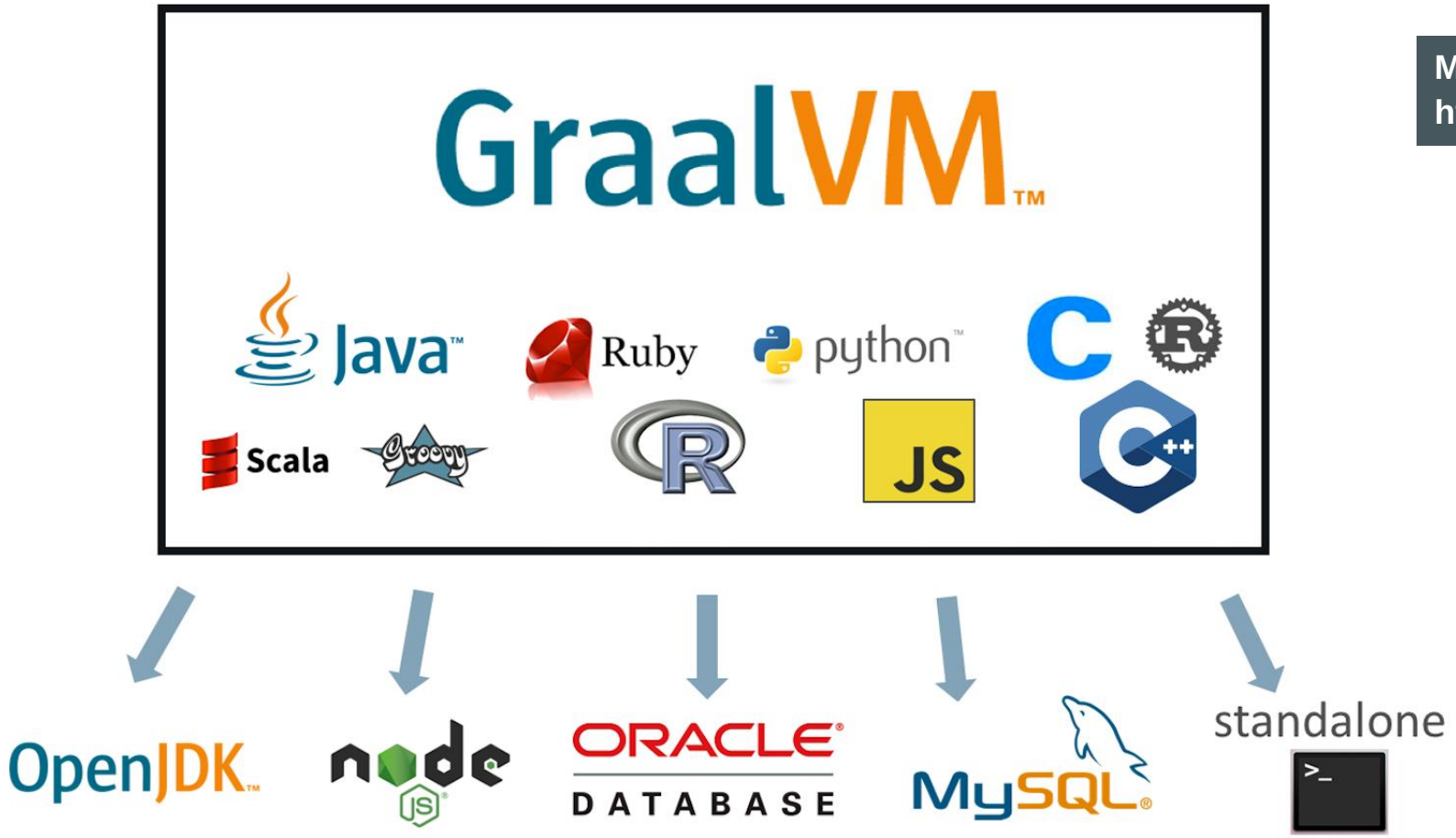
Compiling Scala Faster with GraalVM

Christian Wimmer, Vojin Jovanovic
VM Research Group, Oracle Labs

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. 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 in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

GraalVM: Run Programs Faster Anywhere



More information at:
https://www.graalvm.org

Graal: One Compiler for Managed Languages

- Written in Java
  - Eases development and maintenance
- Modular architecture
  - Configurable compiler phases
  - Compiler-VM separation: snippets, provider interfaces
- Designed for speculative optimizations and deoptimization
  - Metadata for deoptimization is propagated through all optimization phases
- Designed for exact garbage collection
  - Read/write barriers, pointer maps for garbage collector
- Aggressive high-level optimizations

Graal: Partial Escape Analysis (1)

```
public static Car getCached(int hp, String name) {
        Car car = new Car(hp, name, null);
        Car cacheEntry = null;
        for (int i = 0; i < cache.length; i++) {
                if (car.hp == cache[i].hp &&
                          car.name == cache[i].name) {
                        cacheEntry = cache[i];
                        break;
                }
        }
        if (cacheEntry != null) {
                return cacheEntry;
        } else {

                addToCache(car);
                return car;
        }
}
```

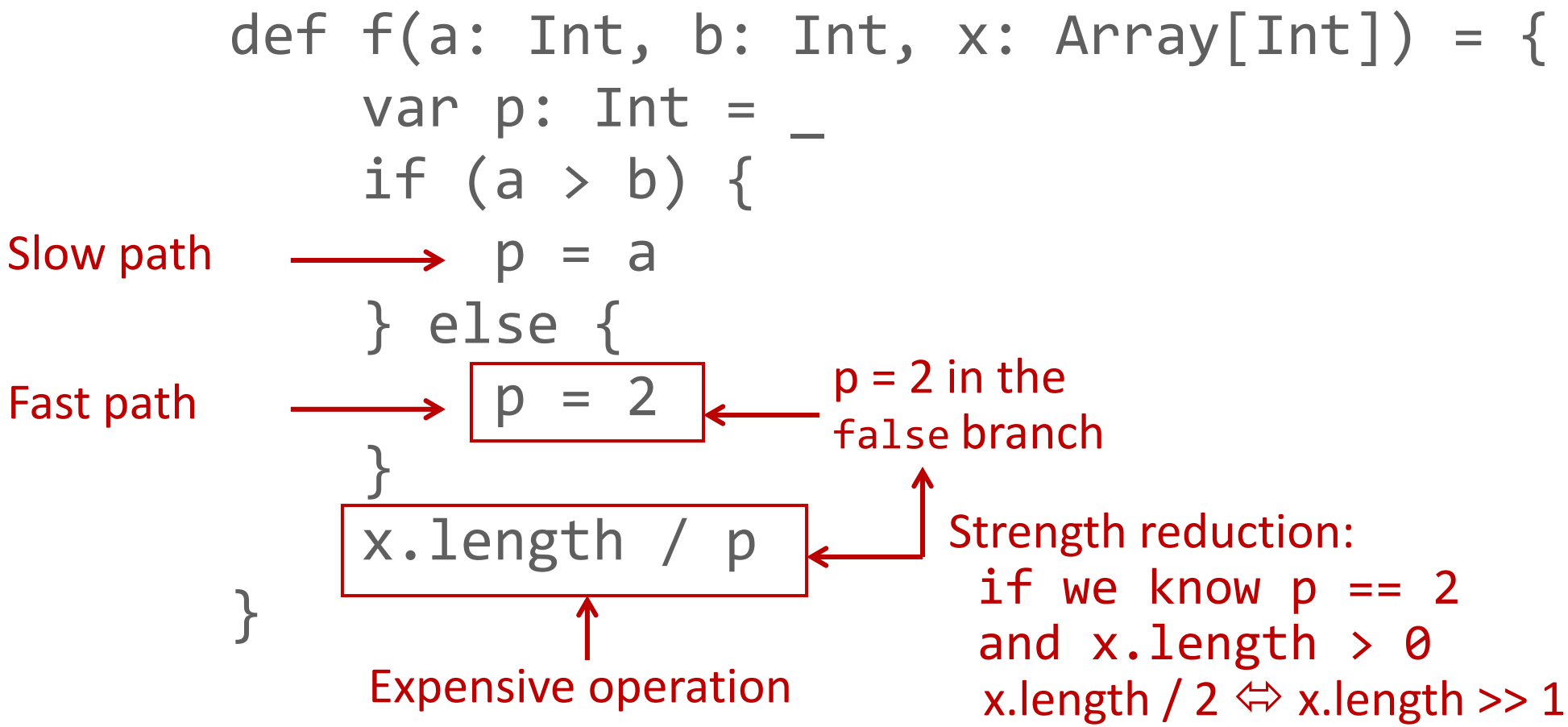
Graal: Partial Escape Analysis (2)

```
public static Car getCached(int hp, String name) {

    Car cacheEntry = null;
    for (int i = 0; i < cache.length; i++) {
        if (hp == cache[i].hp &&
              name == cache[i].name) {
            cacheEntry = cache[i];
            break;
        }
    }
    if (cacheEntry != null) {
        return cacheEntry;
    } else {
        Car car = new Car(hp, name, null);
        addToCache(car);
        return car;
    }
}
```

- **new** Car(...) escapes at:
  - addToCache(car);
  - **return** car;
- Might be a very unlikely path
- No allocation in frequent path

Graal: Simulation Based Path Duplication (1)



Graal: Simulation Based Path Duplication (2)

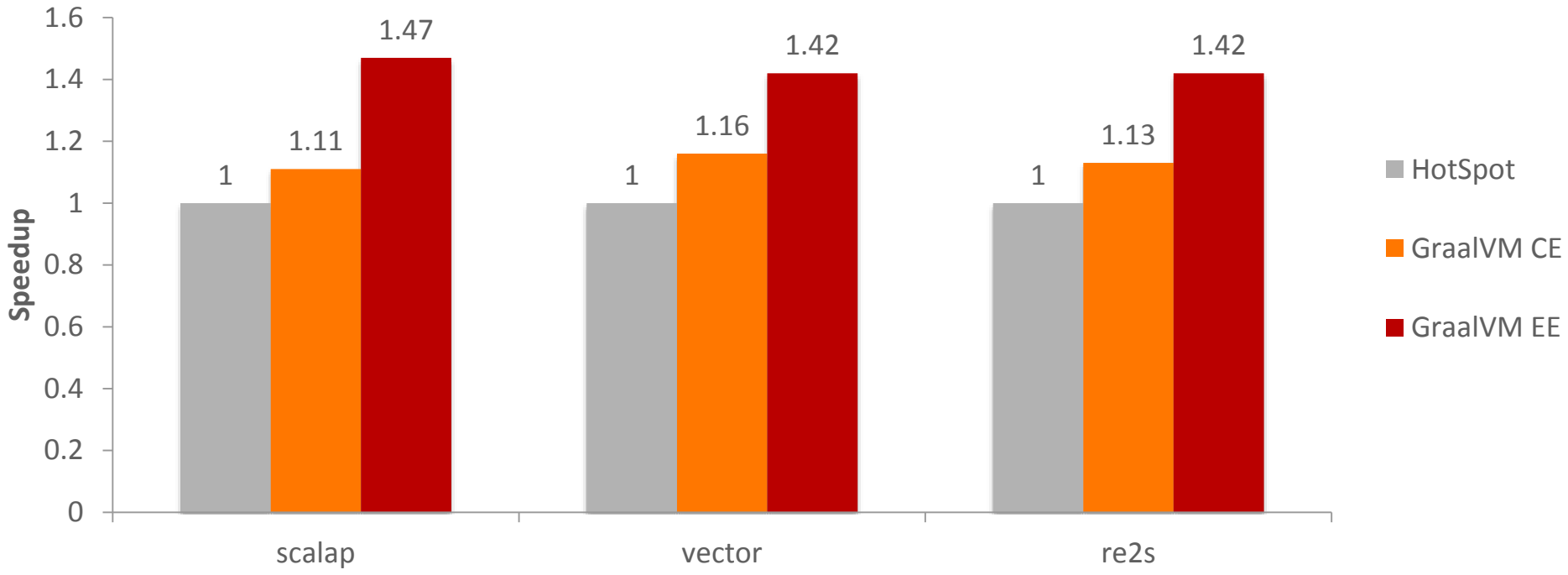
- Cost-benefit analysis for duplication based on a cost model
  - Benefit: (Latency(Div) - Latency(Shift)) * Probability = 31 * 0.9 = 27.9
  - Cost: 5 instructions
    - Add 1 Additional Return (+ 4 Instructions) + 1 Additional Shift + 1 Additional Read
    - Subtract 1 Jump from branch

```
def f(a: Int, b: Int, x: Array[Int]) =
    if (a > b) {
        x.length / a;
    } else {
        x.length >> 1;
    }
```

One order of magnitude faster on Intel CPUs

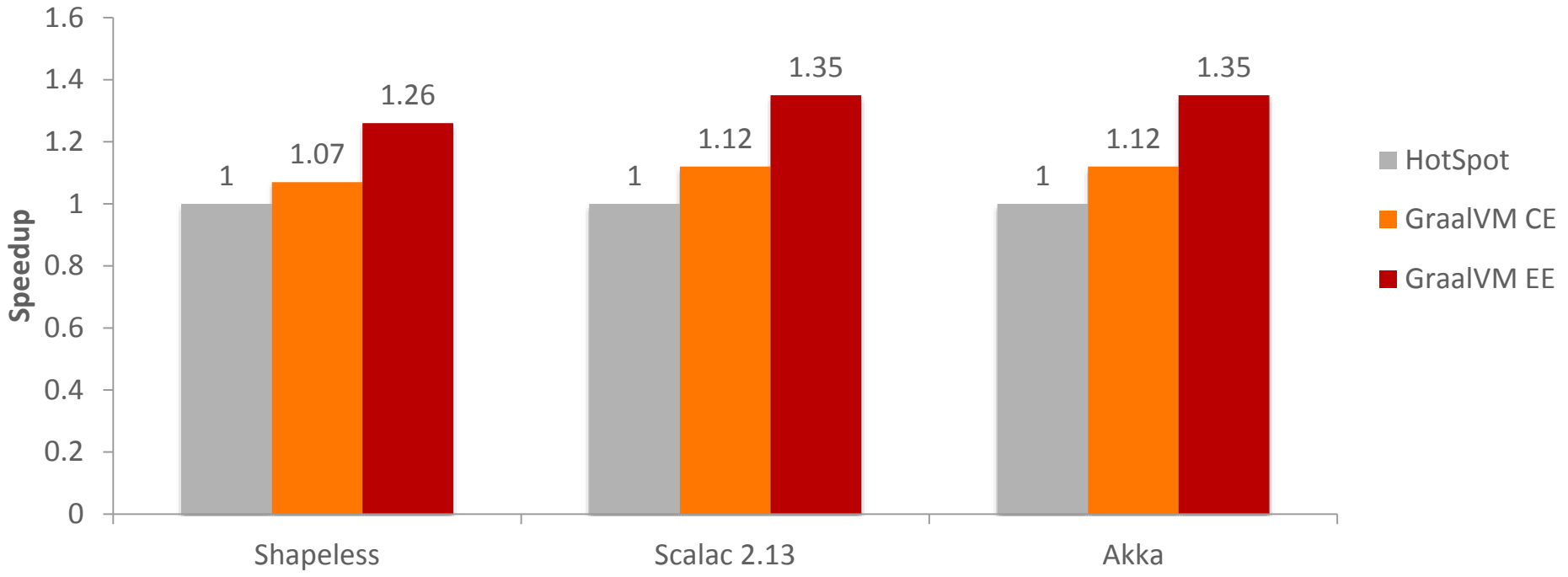
Benchmarks: The Scala 2.12.6 Compiler

- Benchmarks from the Scala compiler benchmark suite



Benchmarks: Building Scala Projects with SBT

- `sbt compile` step with a warmed up VM



Using GraalVM for Scala

- GraalVM comes in two versions
  - GraalVM CE can be used freely by anyone
  - GraalVM EE free for evaluation and non-production usage

- GraalVM EE can be downloaded from

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

- To enable GraalVM for Scala compilation:

```
sbt --java-home <path-to-graal-vm>
```

Managed Runtimes: Slow Startup and High Footprint

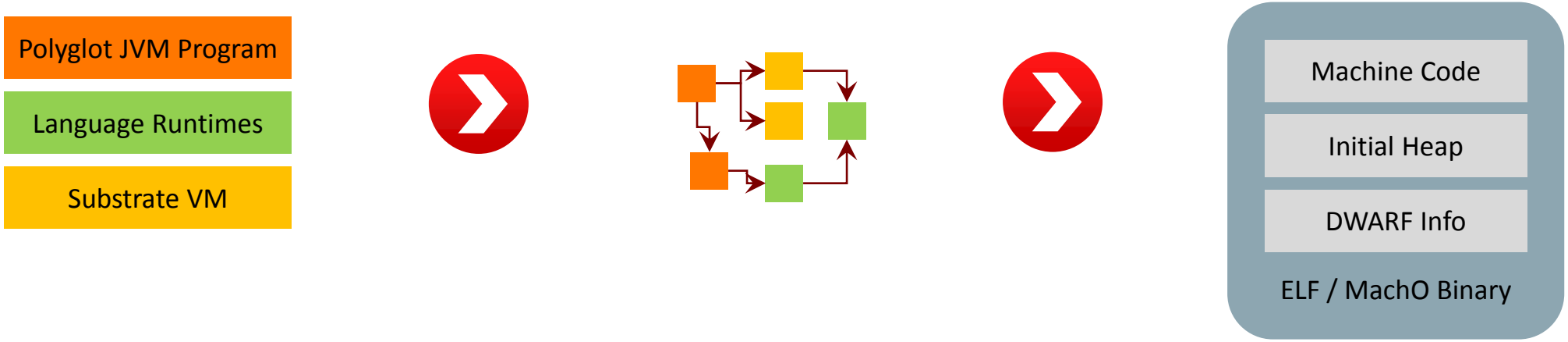
- Slow startup and high footprint comes from
  - Class loading
  - Bytecode interpretation or baseline compilation
  - Just-in-time compilation

Program	Time	Instructions	Memory
“Hello, World!” in C	0.005s	154,127	450 KByte
“Hello, World!” in Scala	0.109s	162,673,275	25,000 KByte
“Hello, World!” in JS on the JVM	1.268s	3,272,118,178	120,000 KByte

Native Image: Execution Model

Points-To Analysis

Ahead-of-Time Compilation



All classes from
the user application,
all language runtimes,
and Substrate VM

Reachable methods,
fields, and classes

Application or shared library running
without dependency on JDK
and without Java class loading

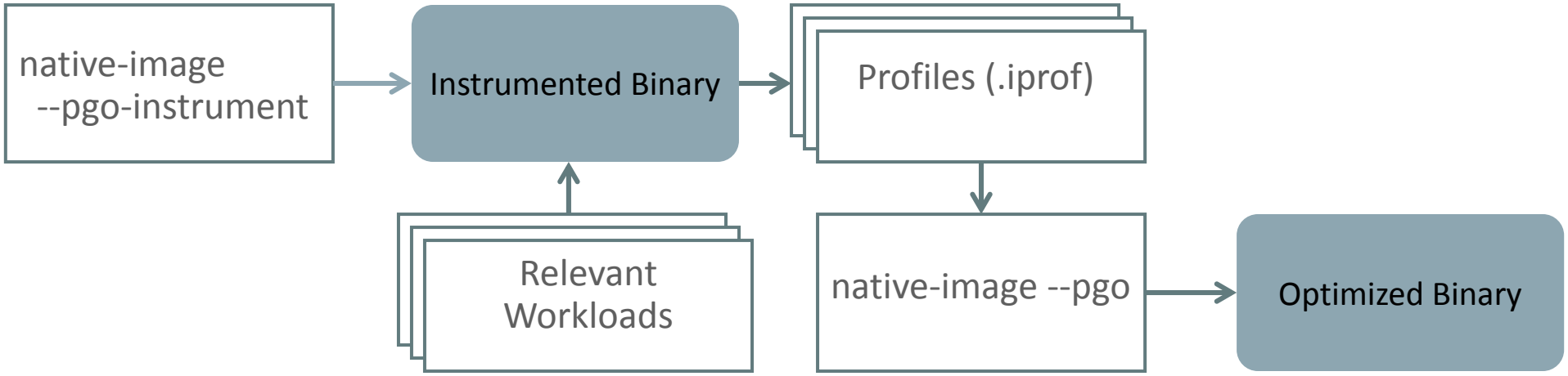
Native Image: Startup

- Bytecode compiled with Graal in AOT mode
  - produced assembly contains no profiling code, only application logic
- Startup performance comparable to C

Program	Time	Instructions	Memory
“Hello, World!” in C	0.005s	154,127	450 KByte
“Hello, World!” in Scala (Native Image)	0.006s	232,122	780 KByte
“Hello, World!” in JS (Native Image)	0.028s	520,000	3,900 KByte

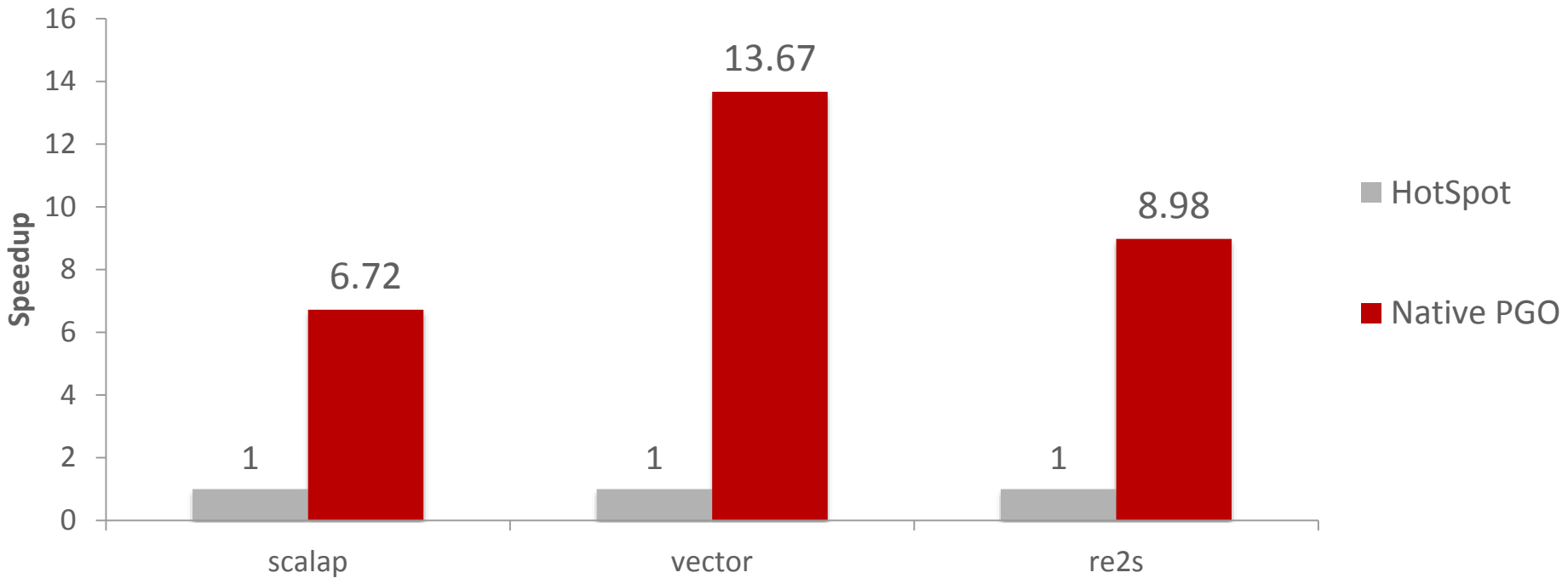
Native Image: Profile-Guided Optimizations (PGO)

- Graal compiler is built ground-up with profiles in mind
  - Collecting profiles is essential for performance of native images
- PGO requires running relevant workloads before building an image



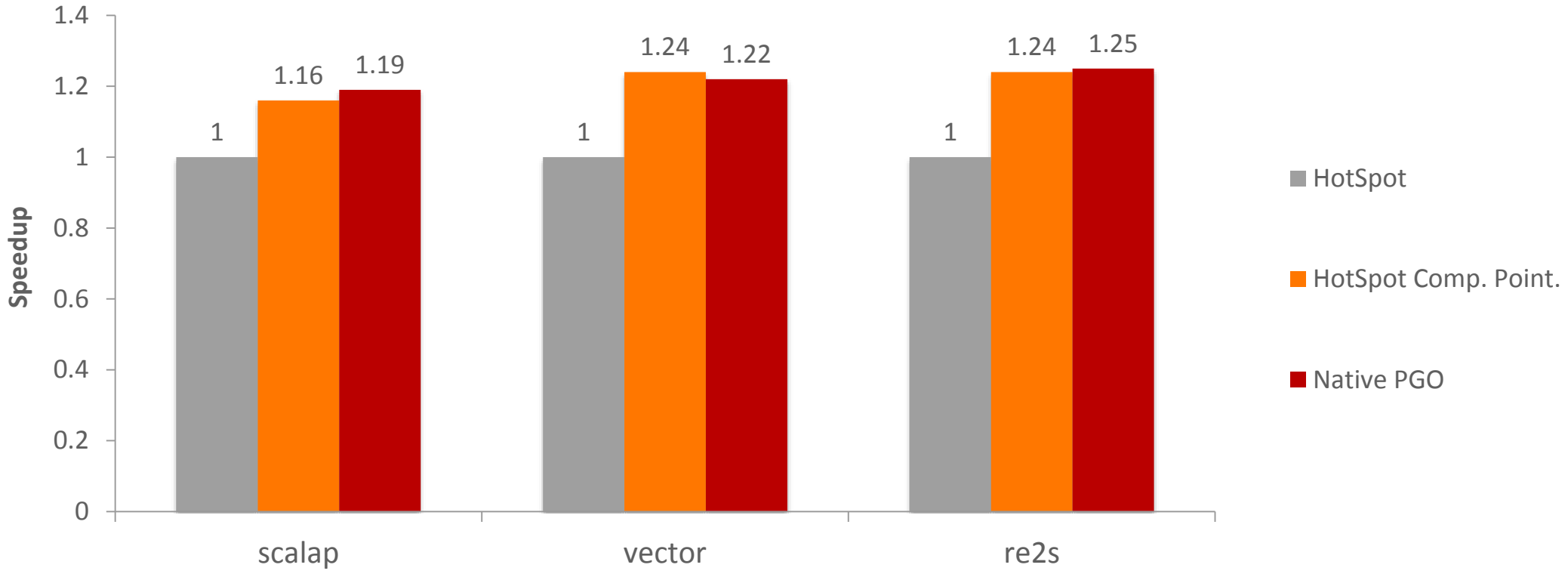
Native Image: Performance of Cold Scala Compilation

- Benchmarks from the Scala compiler benchmark suite



Native Image: Performance of Hot Scala Compilation

- Benchmarks from the Scala compiler benchmark suite



Native Image: Limitations

- Compiled programs must be known ahead of time
  - Dynamically loaded classes must be known during image build
  - `invokedynamic` will not work in general
  - No bytecode generation at runtime
- Currently not implemented
  - Parts of the JDK, e.g., Swing and AWT

What about Scala Macros?

- Scala macros use dynamic class loading
  - Dynamically loaded by scalac
  - Not always known at image build time
- Current solution
  - Build a custom native executable for a project with all macros included

Native Image: C Interoperability for Java

```
@CFunction static native int clock_gettime(int clock_id, timespec tp);
```

```
int clock_gettime(clockid_t __clock_id, struct timespec *__tp)
```

```
@CConstant static native int CLOCK_MONOTONIC();
```

```
#define CLOCK_MONOTONIC 1
```

```
@CStruct interface timespec extends PointerBase {
  @CField long tv_sec();
  @CField long tv_nsec();
}
```

```
struct timespec {
  __time_t tv_sec;
  __syscall_slong_t tv_nsec;
};
```

```
@CPointerTo(nameOfCType="int") interface CIntPointer extends PointerBase {
  int read();
  void write(int value);
}
```

```
int* pint;
```

```
@CPointerTo(CIntPointer.class) interface CIntPointerPointer ...
```

```
int** ppint;
```

```
@CContext(PosixDirectives.class)
```

```
#include <time.h>
```

```
@CLibrary("rt")
```

```
-lrt
```

Implementation of `System.nanoTime()` using with C interop:

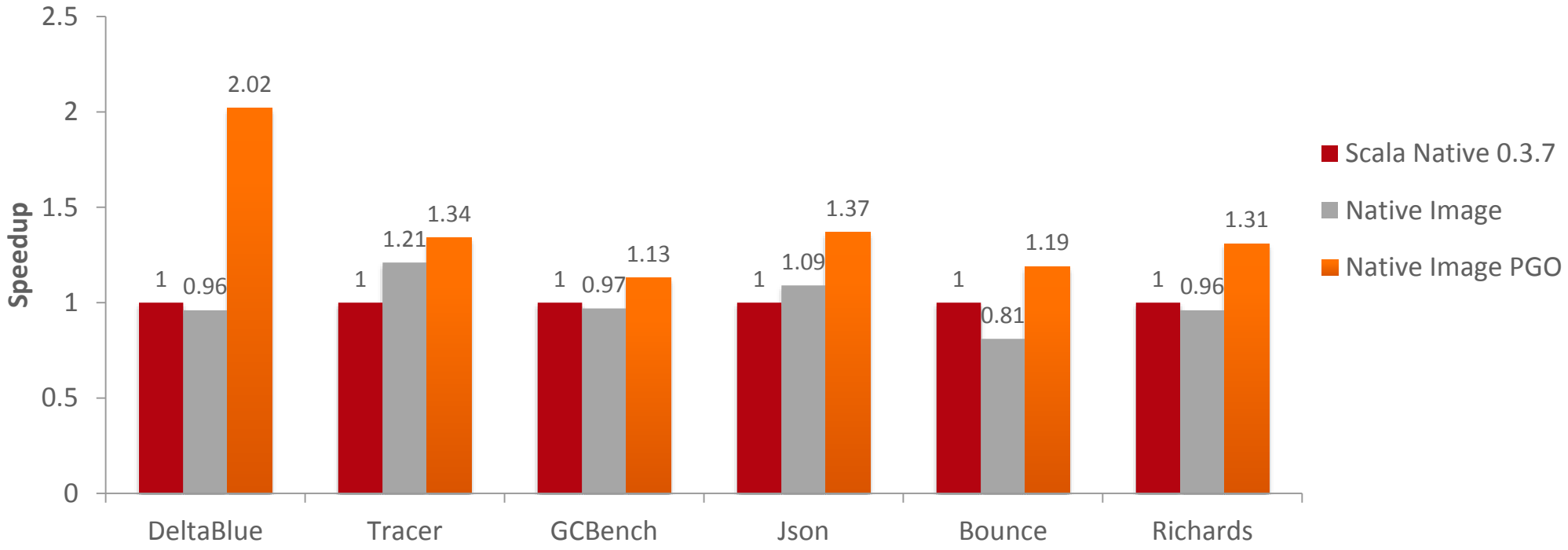
```
static long nanoTime() {
  timespec tp = StackValue.get(SizeOf.get(timespec.class));
  clock_gettime(CLOCK_MONOTONIC(), tp);
  return tp.tv_sec() * 1_000_000_000L + tp.tv_nsec();
}
```

Scala Native via Native Image

- Scala Native provides an idiomatic interface
- Scala Native can be implemented via GraalVM
  - Translate Scala Native intrinsics into GraalVM intrinsics
  - Can be implemented as a simple compiler plugin
- All JVM libraries would become available with GraalVM
  - No need to re-write parts of the JVM ecosystem

Performance of Scala Native vs. Native Image

- Scala Native Benchmarks running with the immix GC



GraalVM: Run Programs Faster Anywhere

- Scala compilation with GraalVM 1.3x-1.5x faster
- Native images of Scala programs
  - Fast startup and low footprint
  - Faster than HotSpot JIT compiled code
- Try it today
  - https://www.graalvm.org
- Demos available on GitHub
  - https://github.com/graalvm/graalvm-demos/scala-days-2018
- Blog article explaining the demo and the results
  - https://medium.com/graalvm

Integrated Cloud

Applications & Platform Services


ORACLE®