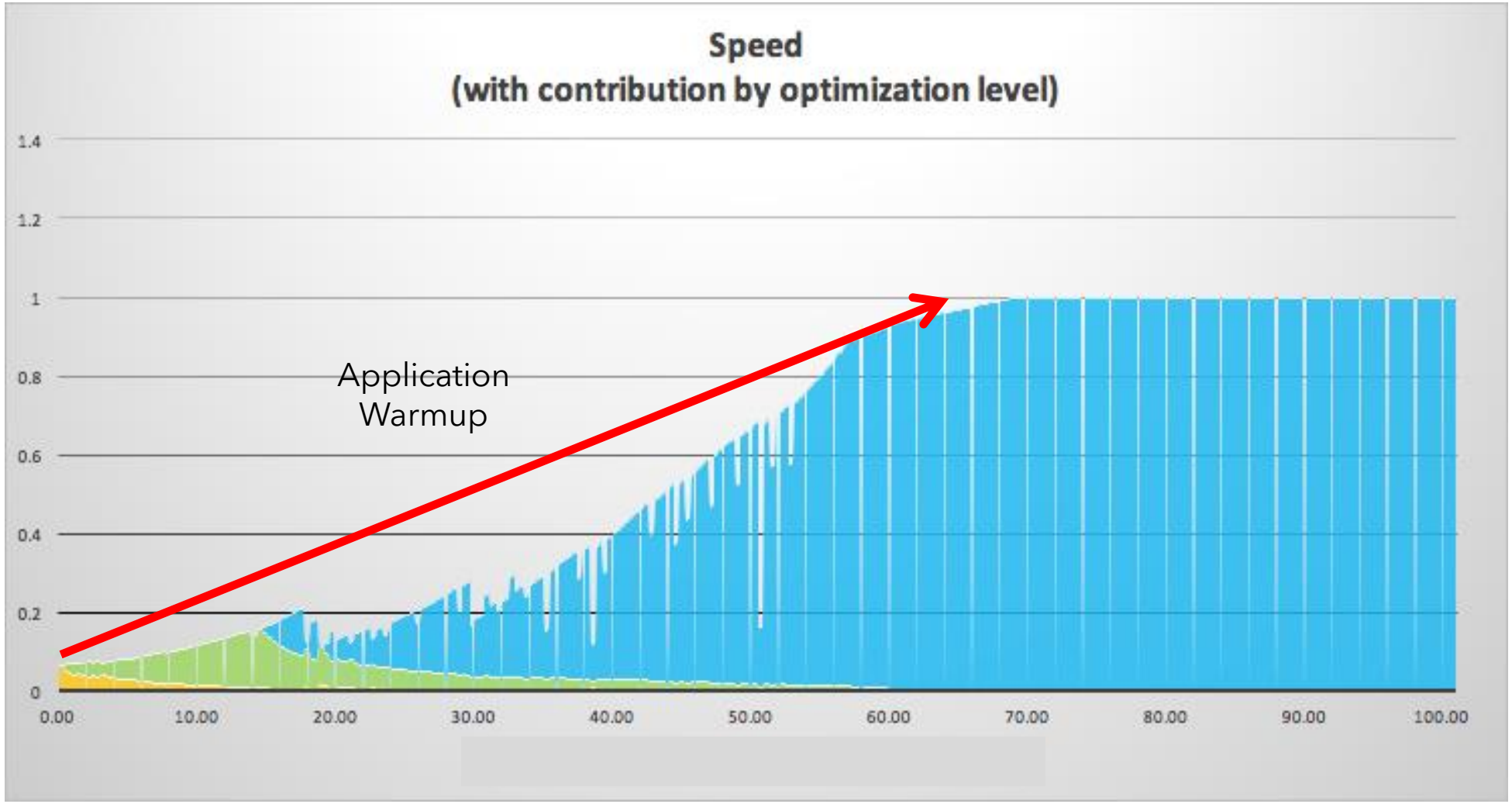
Microsoft

Instant Coffee: How To Eliminate Java Performance Warmup

Simon Ritter, Deputy CTO, Azul

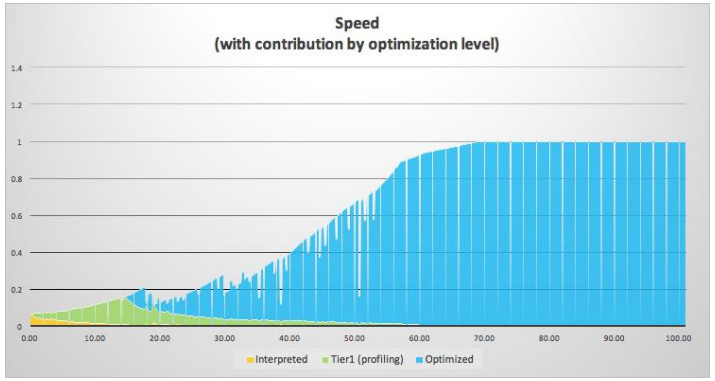
Code. Cloud. Community.

JVM Performance Graph

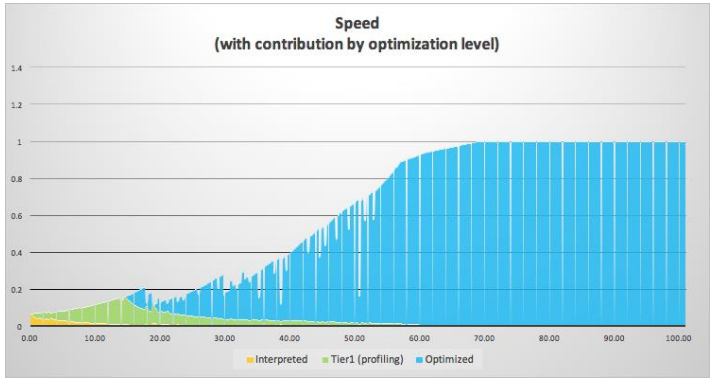


JVM Performance Graph

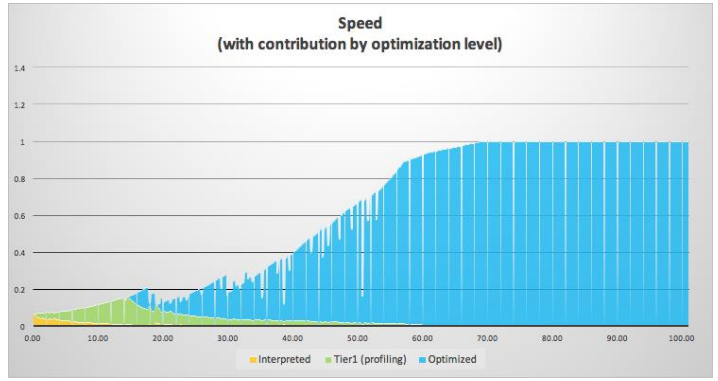
First run



Second run

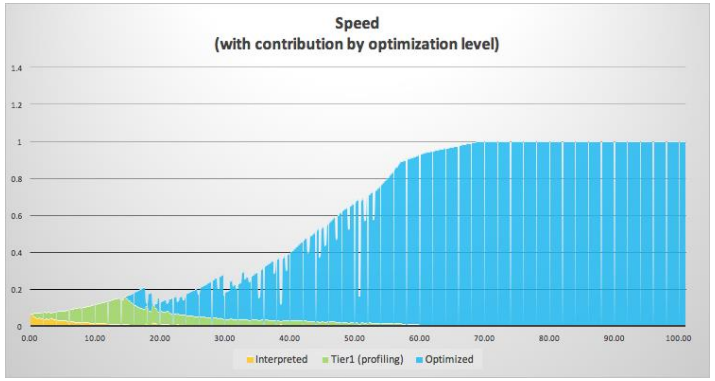


Third run

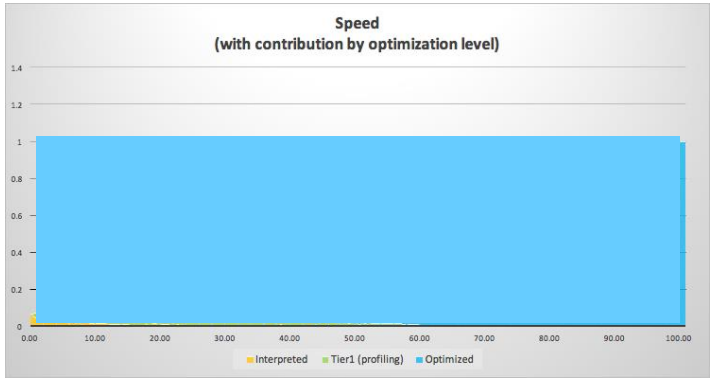


JVM Performance Graph

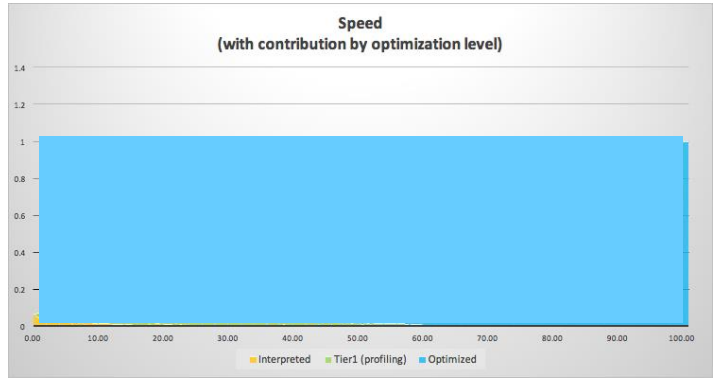
First run



Second run



Third run



azul

Solution 1:
Ahead of Time (AOT) Compilation

Compile Java Source Direct to Native Code

- Traditional approach: Ahead of time, static compilation
- No interpreting bytecodes
- No analysis of hotspots
- No runtime compilation of code placing heavy load on CPUs
- Start at full speed, straight away

- This is the Graal native image approach
- Problem solved, right?

Not So Fast

- AOT is, by definition, static
- And code is compiled *before it is run*
- The compiler has no knowledge of how the code will *actually* run
  - Profile guided optimisation has been around for a long time and only helps partially

Speculative Optimisation Example:
Branch Analysis

```
int computeMagnitude(int value) {
  if (value > 9)
    bias = computeBias(value);
  else
    bias = 1;

  return Math.log10(bias + 99);
}
```

Profiling data shows that *value* (so far) has never been greater than 9

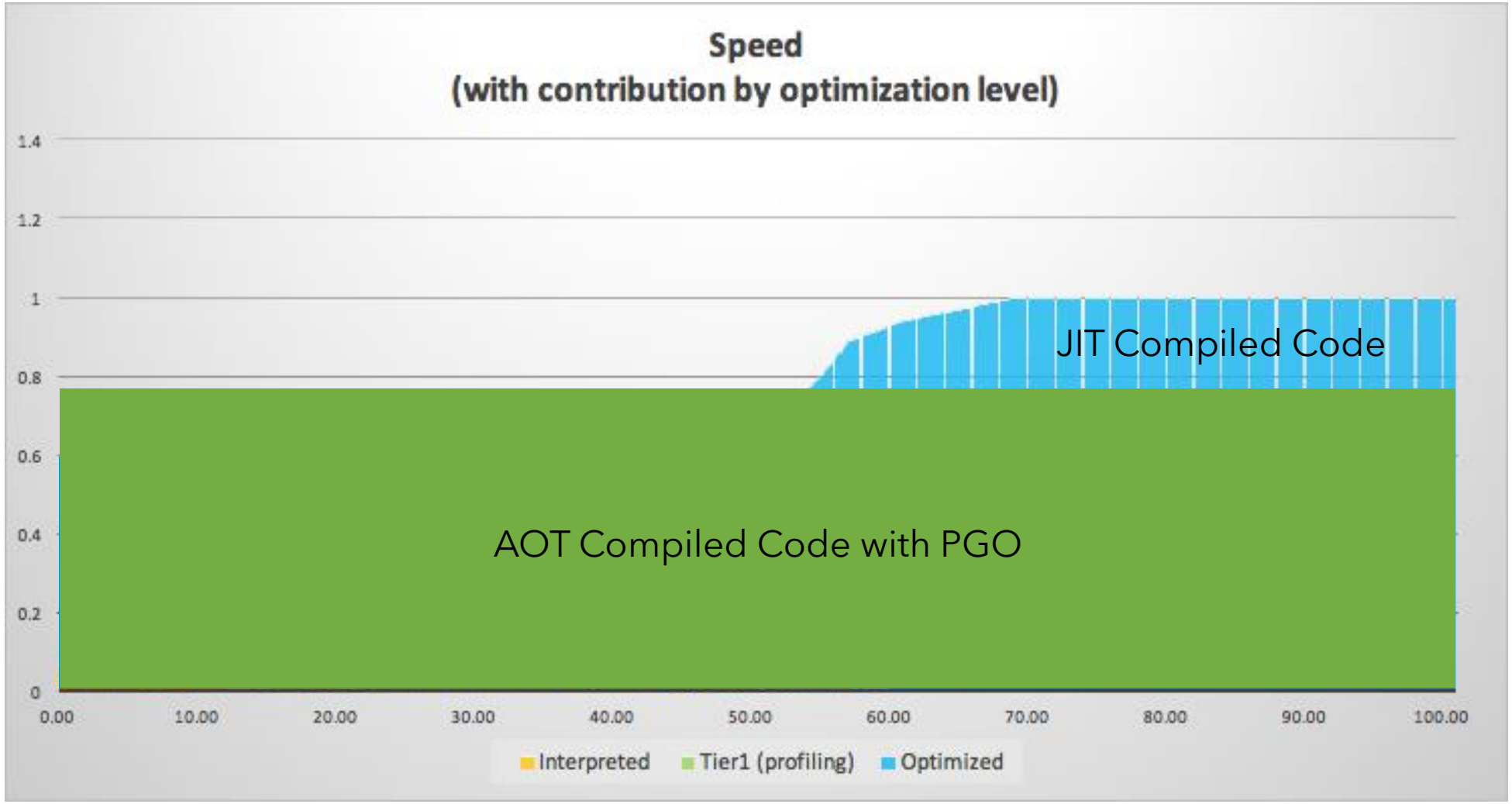
Speculative Optimisation Example:
Branch Analysis

```
int computeMagnitude(int value) {
  if (value > 9)
    uncommonTrap();  // Deoptimise

  return 2;  // Math.log10(100)
}
```

Assume that, based on profiling, *value* will continue to be less than 10

JVM Performance



When To Use AOT

- Ephemeral microservices
  - Startup and warmup time is more important than overall speed
  - Garbage collection is usually a non-issue
- Resource constrained services
  - E.g. 2 vcore container
  - JIT compilation will significantly reduce throughput during warmup

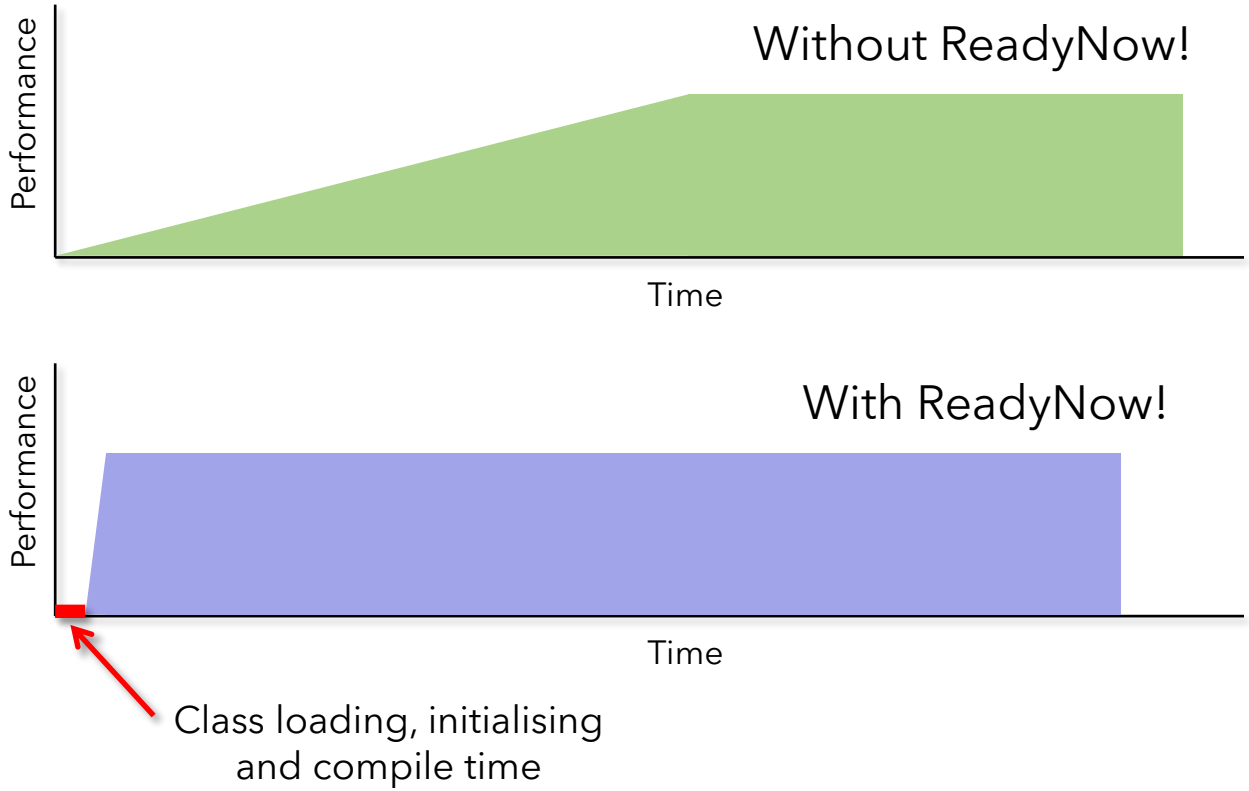
azul

Solution 2:
Store JIT Compilation Data

Azul Prime ReadyNow

- Run the application until its warmed up
- Take a profile
  - All currently loaded classes
  - All currently initialised classes
  - JIT profiling data
  - Deoptimisations that occurred
  - A copy of all compiled code
- Restart application
  - Load and initialise all required classes
  - Load code or compile methods
  - All before `main()`

ReadyNow Startup Time



azul

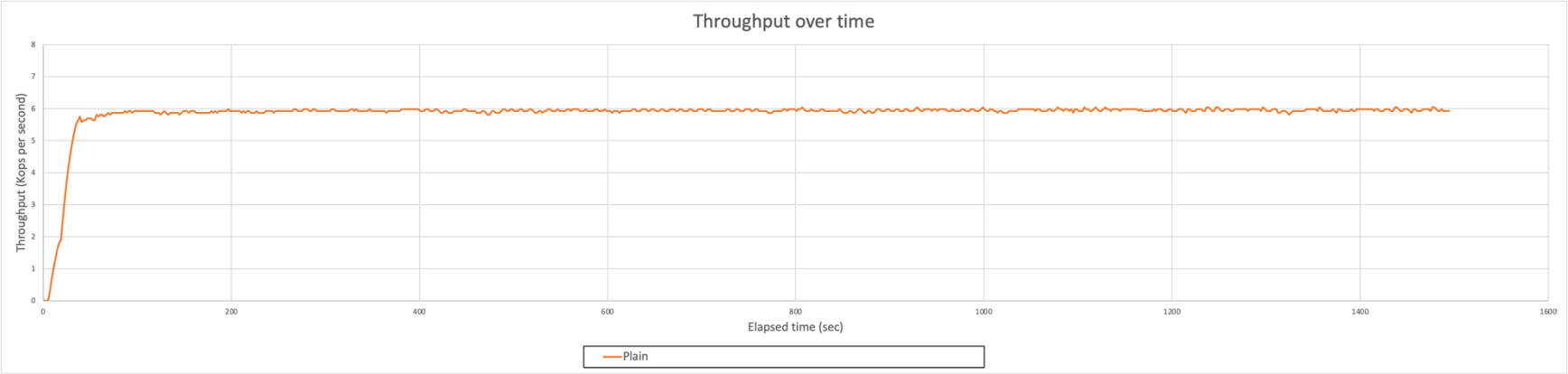
Solution 3:
Decouple The JIT Compiler

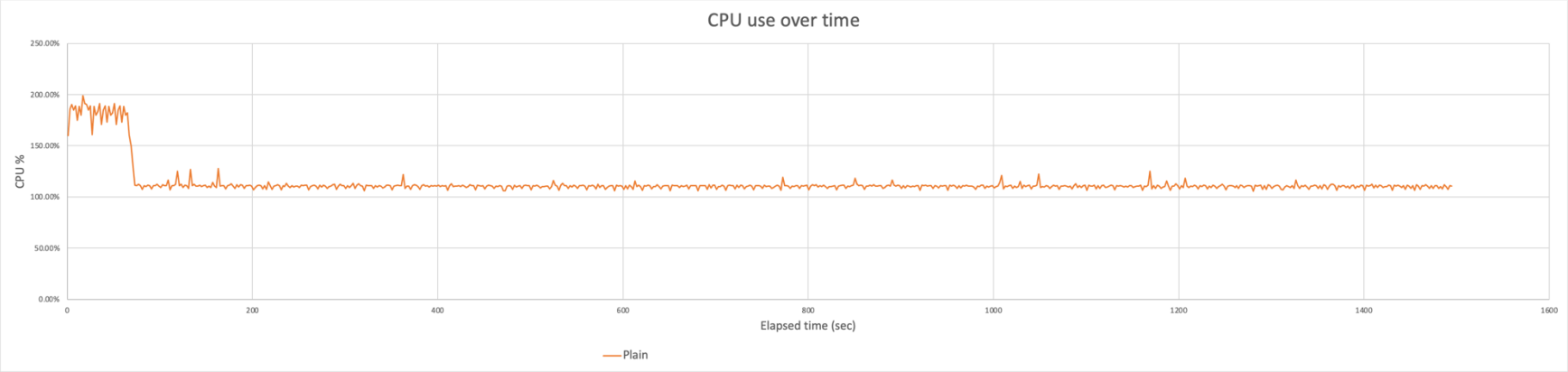
JIT Compilation Has Cost

- JIT is CPU intensive
  - The work has to be done concurrently with the application workload
- Better optimisations deliver better performance (throughput)
  - But require more time, compute power and memory
- This is fine if we have a powerful machine
  - E.g. 64 vcores and 64GB RAM
- Less powerful environments can be problematic
  - E.g. 2vcore container with 2GB RAM
  - Heavily optimised JIT can become prohibitive by degrading throughput
  - Even resul in OOM errors
- Often we end up with a compromise

Speed and CPU Usage Over Time

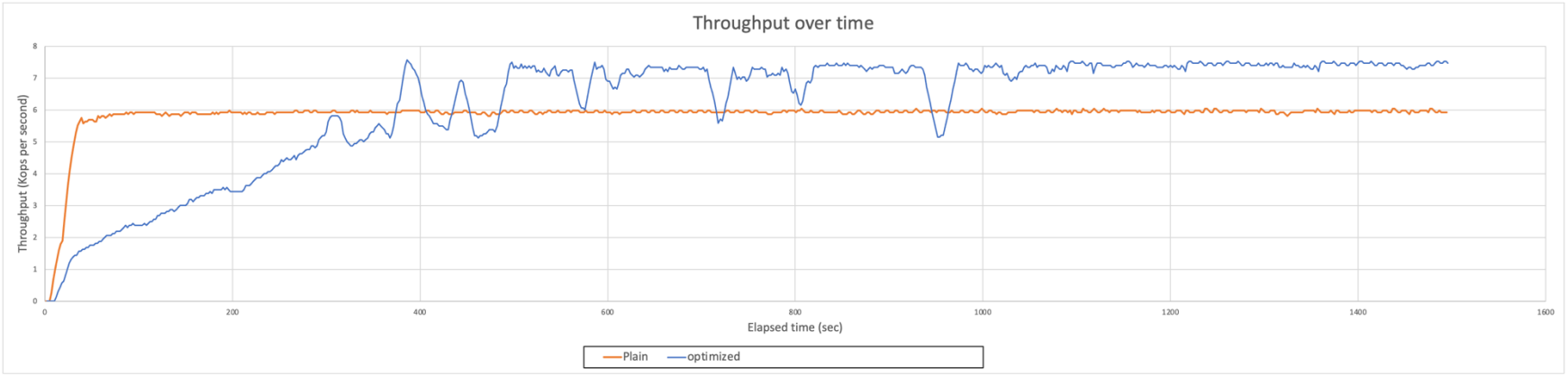
- 2 Vcore container

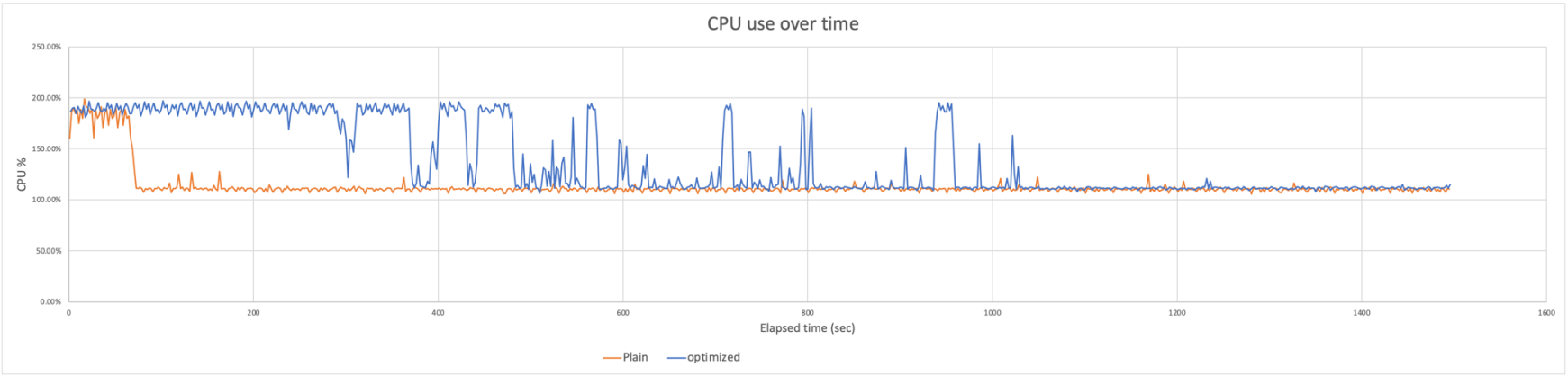




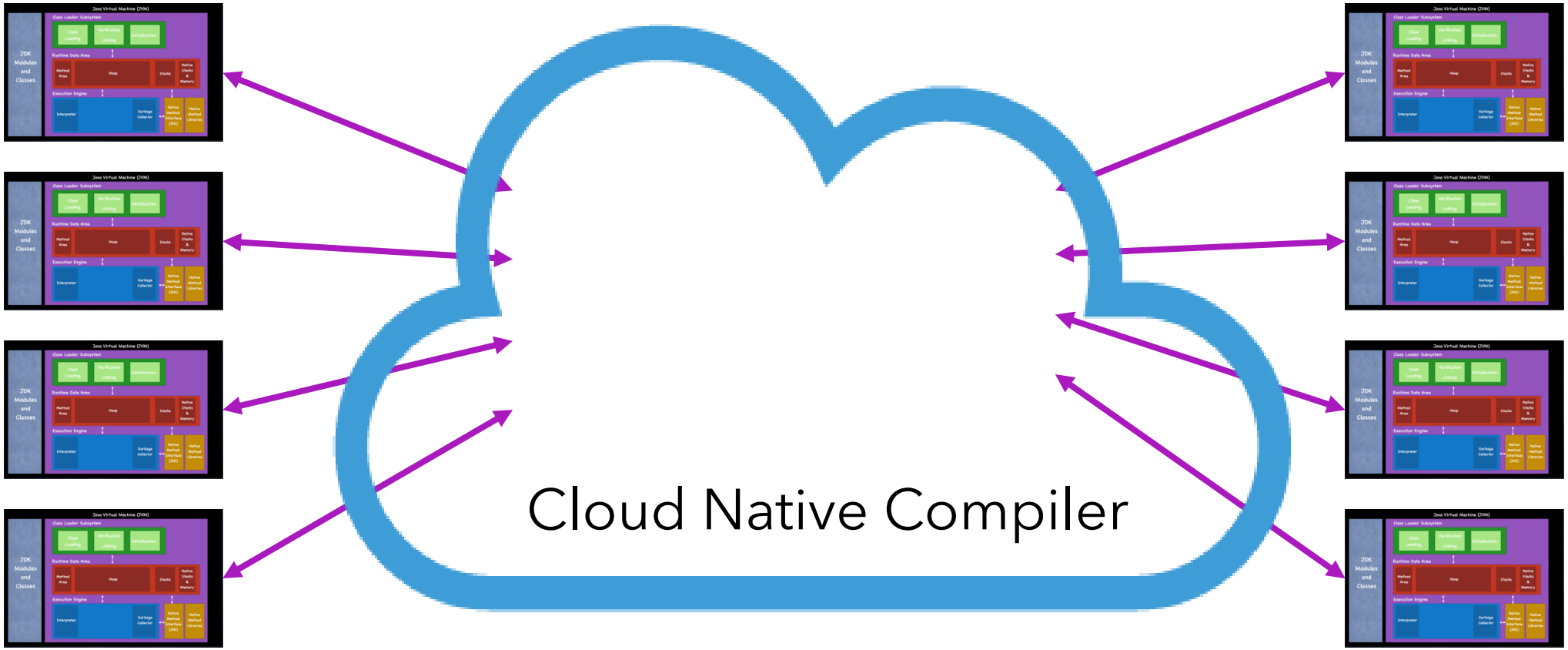
Speed and CPU Usage Over Time

- 2 Vcore container



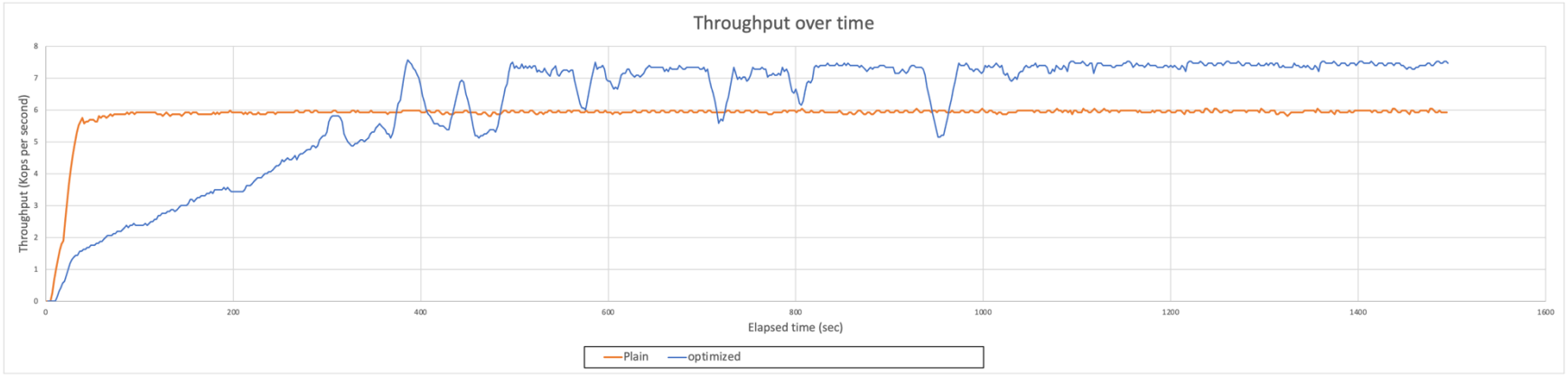


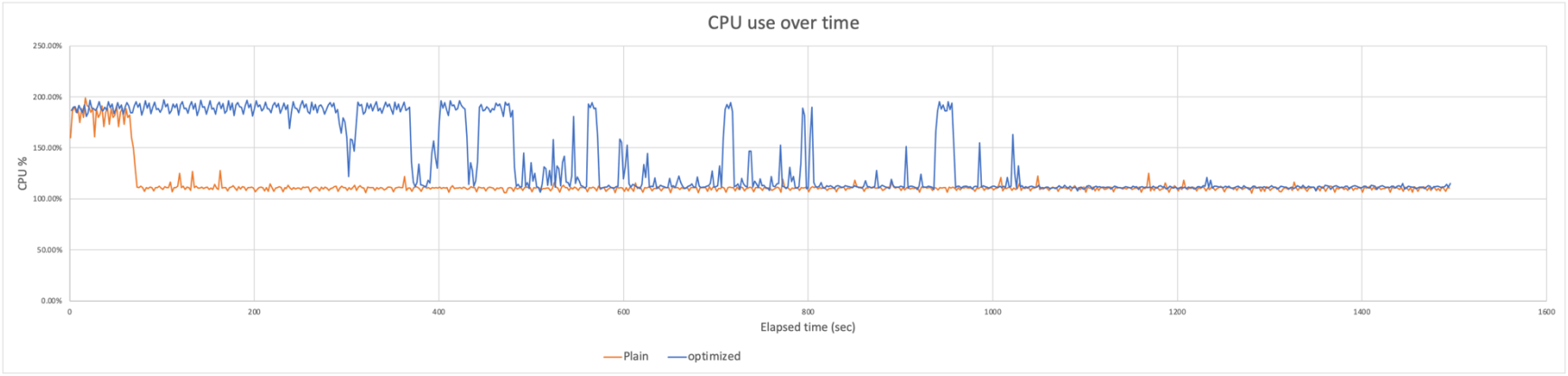
And Made It A Cloud-Based Resource



Speed and CPU Usage Over Time

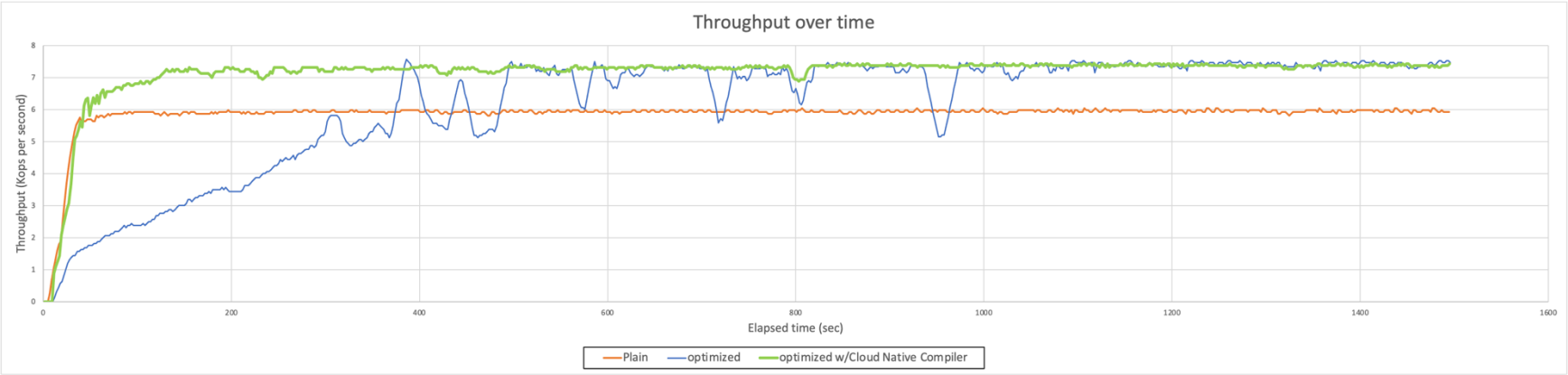
- 2 Vcore container

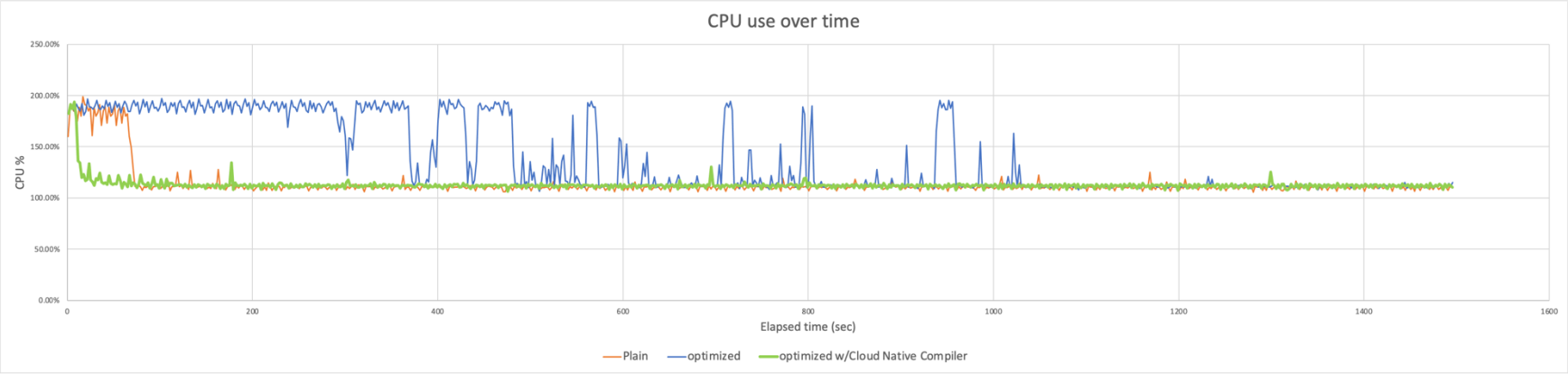




Speed and CPU Usage Over Time

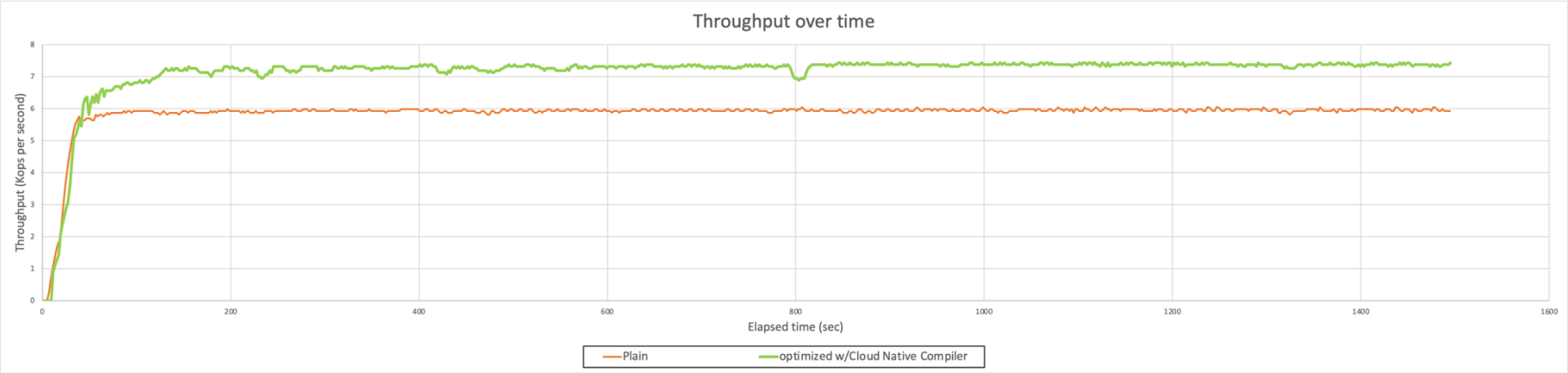
- 2 Vcore container

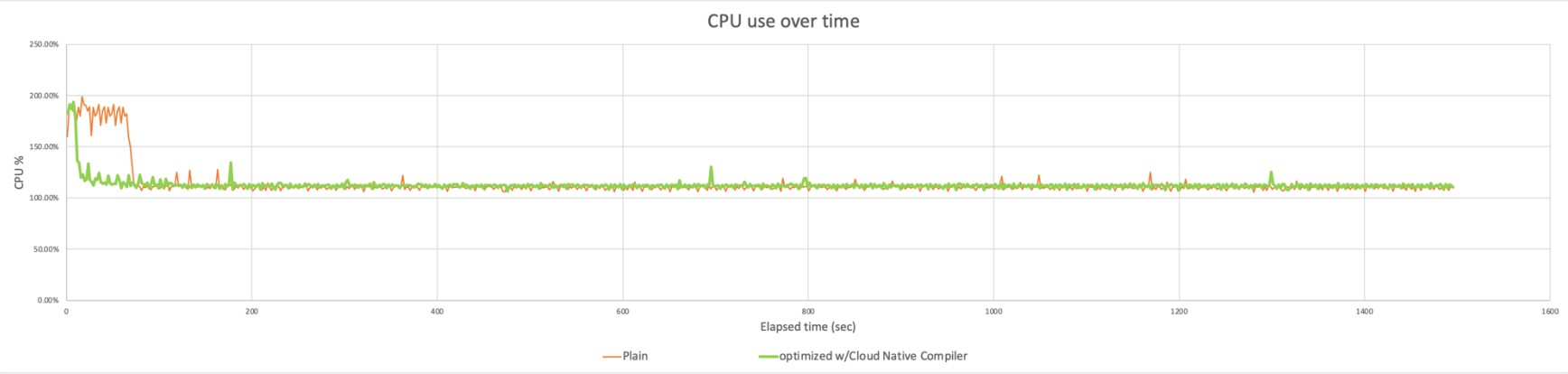




Speed and CPU Usage Over Time

- 2 Vcore container





Isn't This Just Shifting The Cost?

- Well, Yes…
- But we are shifting it to a much more efficient place
- When a JVM optimizes locally, it must carry dedicated resources to do so
- When outsourced to a Cloud Native Compiler
  - The resources are shared and reused
  - The resources can be elastic

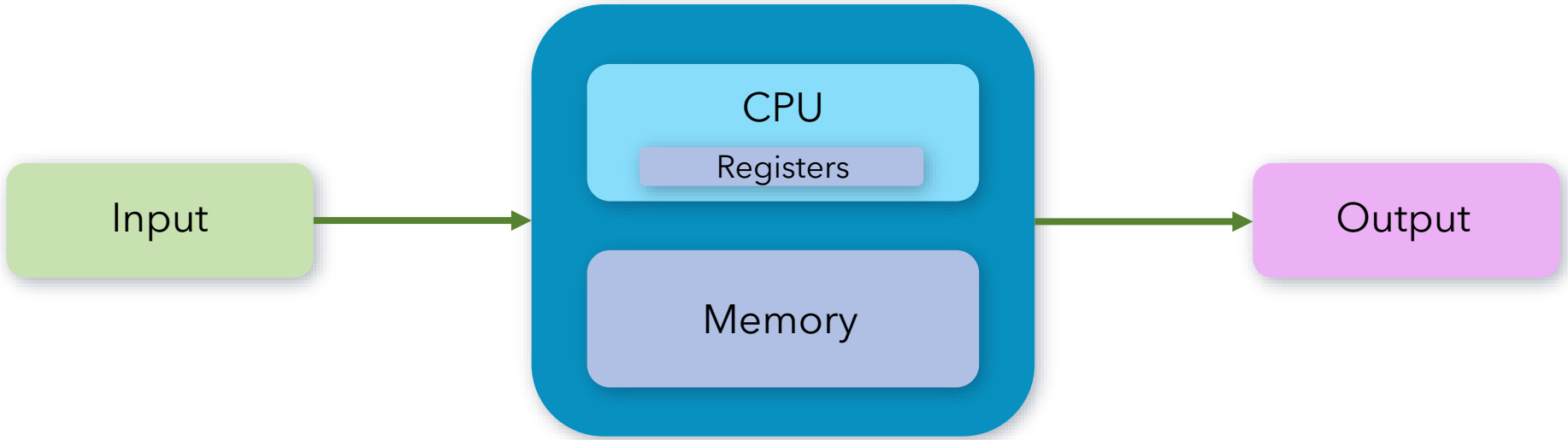
- Compiled code can be cached
  - The JIT now effectively has a memory across runs

azul

Solution 4:
Save The Whole Application State

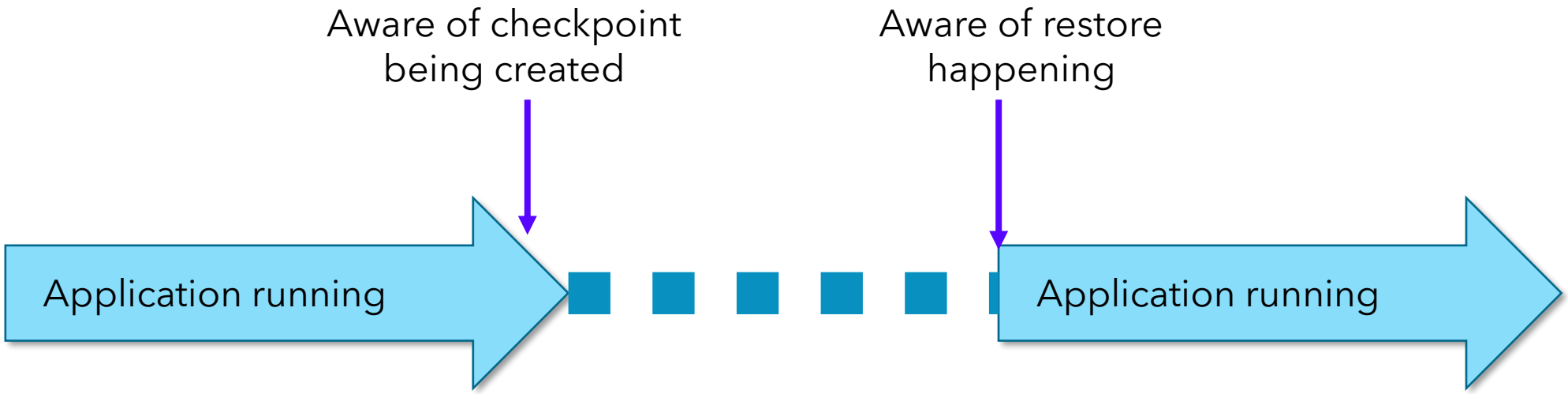
Co-ordinated Resume In Userspace

- Linux project
- Basic idea
  - Freeze a running application
    - Pause program counter
  - Create a snapshot of the applications state (as a set of files)
  - At some later point, use those files to restart the application from the same point
    - Potentially, on a different physical machine



Co-ordinated Restore at Checkpoint (CRaC)

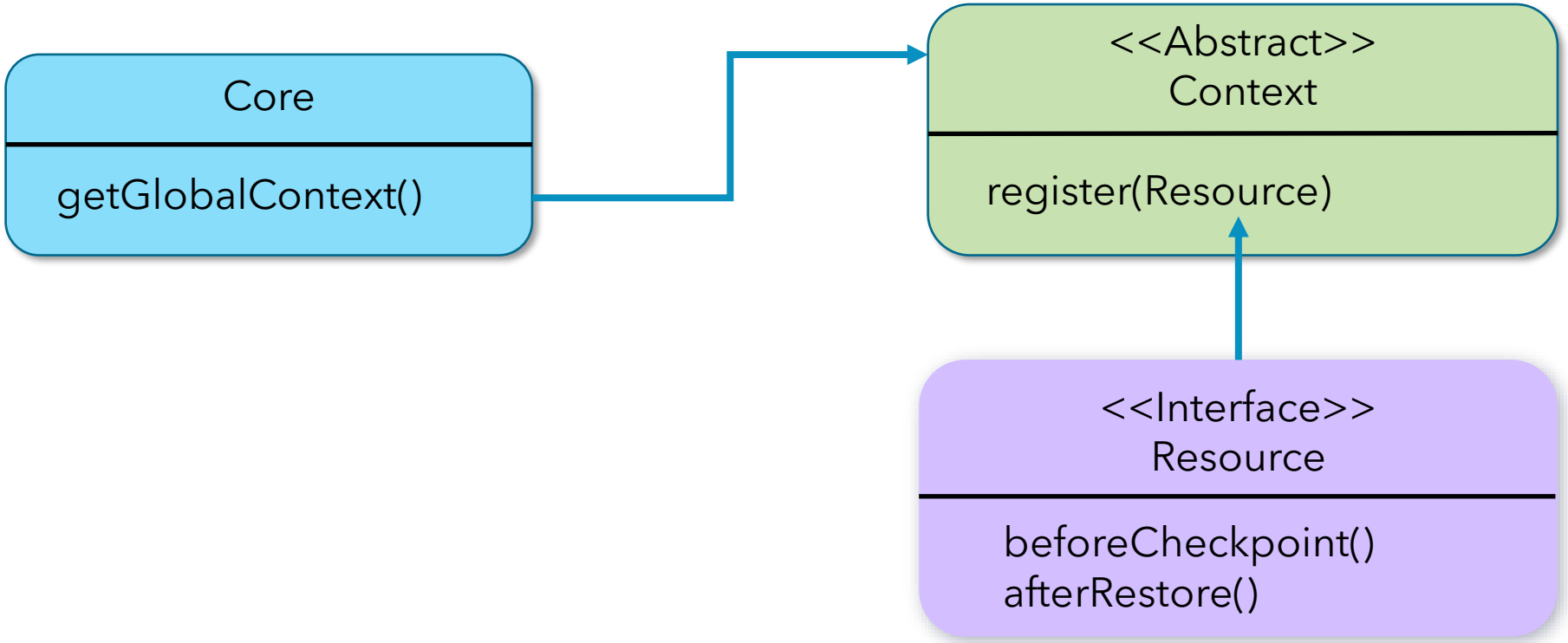
- Let's make the application *aware* it is being checkpointed and restored



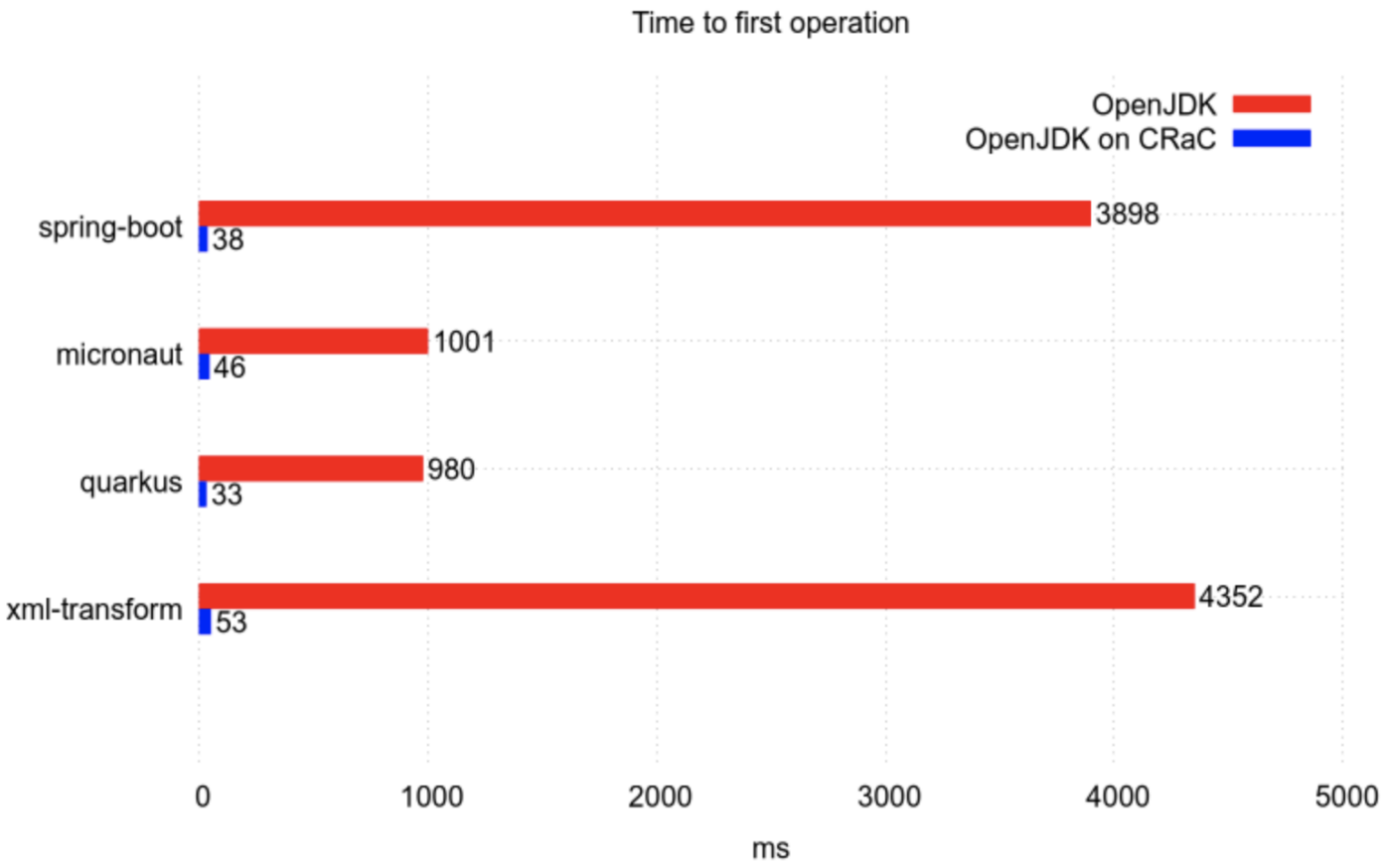
- CRaC also enforces more restrictions on a checkpointed application
  - No open files or sockets
  - Checkpoint will be aborted if any are found

Using CRaC API

- `Resource` objects need to be registered with a `Context` so that they can receive notifications
- There is a global `Context` accessible via the static `getGlobalContext()` method of the `Core` class



Does It Work? POC Results



azul

Summary

Solving The JVM Warmup Problem

- No one solution will fit all situations
- AOT is good for fast startup/small footprint in ephemeral services
- ReadyNow provides memory of JIT across runs
- Cloud Native Compiler offloads JIT workload
- CRaC restarts an application from a known point
- Project Leyden is looking at approaches that include those above as well as other ideas

Thank You.

Simon Ritter, Deputy CTO

@speakjava