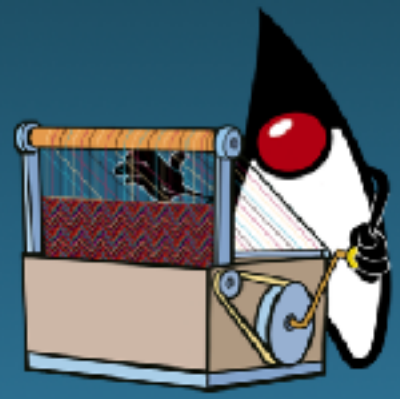
Project Loom

Fibers and Continuations



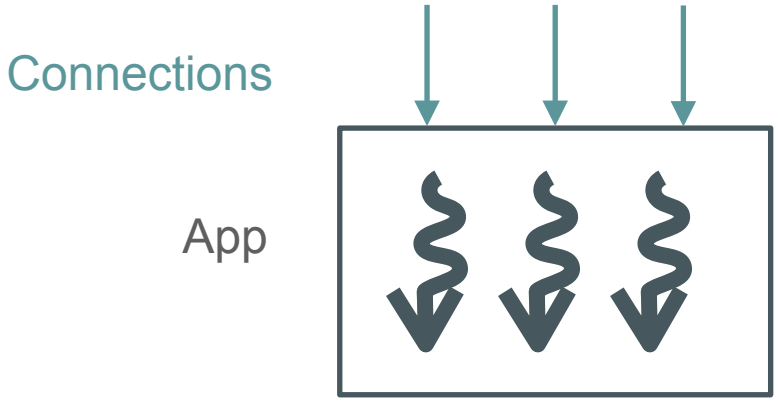
Alan Bateman
Java Platform Group, Oracle
November 2018

Project Loom

- Continuations
- Fibers
- Tail-calls

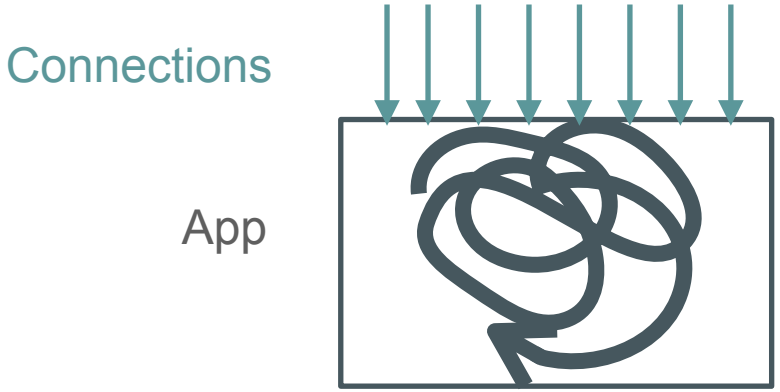
Why Fibers

Today, developers choose between



simple (blocking / synchronous),
but less scalable code (with threads)

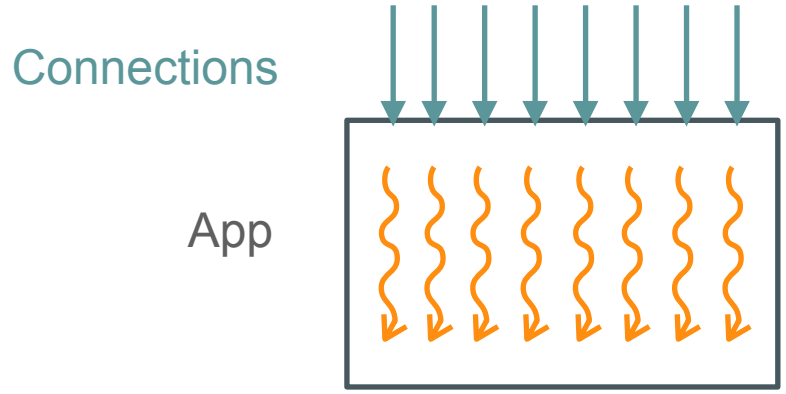
and



complex, non-legacy-interoperable,
but scalable code (asynchronous)

Why Fibers

With fibers, devs have *both*: simple, familiar, maintainable, interoperable code, that is also scalable



Fibers make even existing server applications consume fewer machines (by increasing utilization), significantly reducing costs

Continuations

A continuation (precisely: delimited continuation) is a program object representing a computation that may be suspended and resumed (also, possibly, cloned or even serialized).

Prototype Continuation API

```
package java.lang;

public class Continuation implements Runnable {

    public Continuation(ContinuationScope scope, Runnable target)

    public final void run()

    public static void yield(ContinuationScope scope)

    public boolean isDone()

    :

}
```

Fibers

A Fiber *light weight* or *user mode thread*, scheduled by the Java virtual machine, not the operating system

Fibers are low footprint and have negilgible task-switching overhead. You can have millions of them!

Why fibers?

- The runtime is well positioned to manage and schedule application threads, esp. if they interleave computation and I/O and interact very often (exactly how server threads behave)

- Make concurrency simple again

fiber = continuation + scheduler

fiber = continuation + scheduler

- A fiber wraps a task in a continuation
  - The continuation yields when the task needs to block
  - The continuation is continued when the task is ready to continue
- Scheduler executes tasks on a pool of *carrier* threads
  - java.util.concurrent.Executor in the current prototype
  - Default/built-in scheduler is a ForkJoinPool

Fiber prototype

- Focus to date has been on the control flow and concepts, not the API
- Minimal `java.lang.Fiber` in current prototype that supports scheduling, park/unpark, and waiting for a fiber to terminate
- java.util.concurrent APIs can park/unpark fibers
- Socket and pipe APIs park fiber rather than block threads in syscalls

How much existing code can fibers run?

- A big question, lots of trade-offs
  - Do we completely re-imagine threads?
  - Can we run all existing code in the context of a fiber?
- Likely to wrestle with these questions for a long time
- Current prototype can run existing code but with some limitations

Example using existing code/libraries

- Example uses Jetty and Jersey

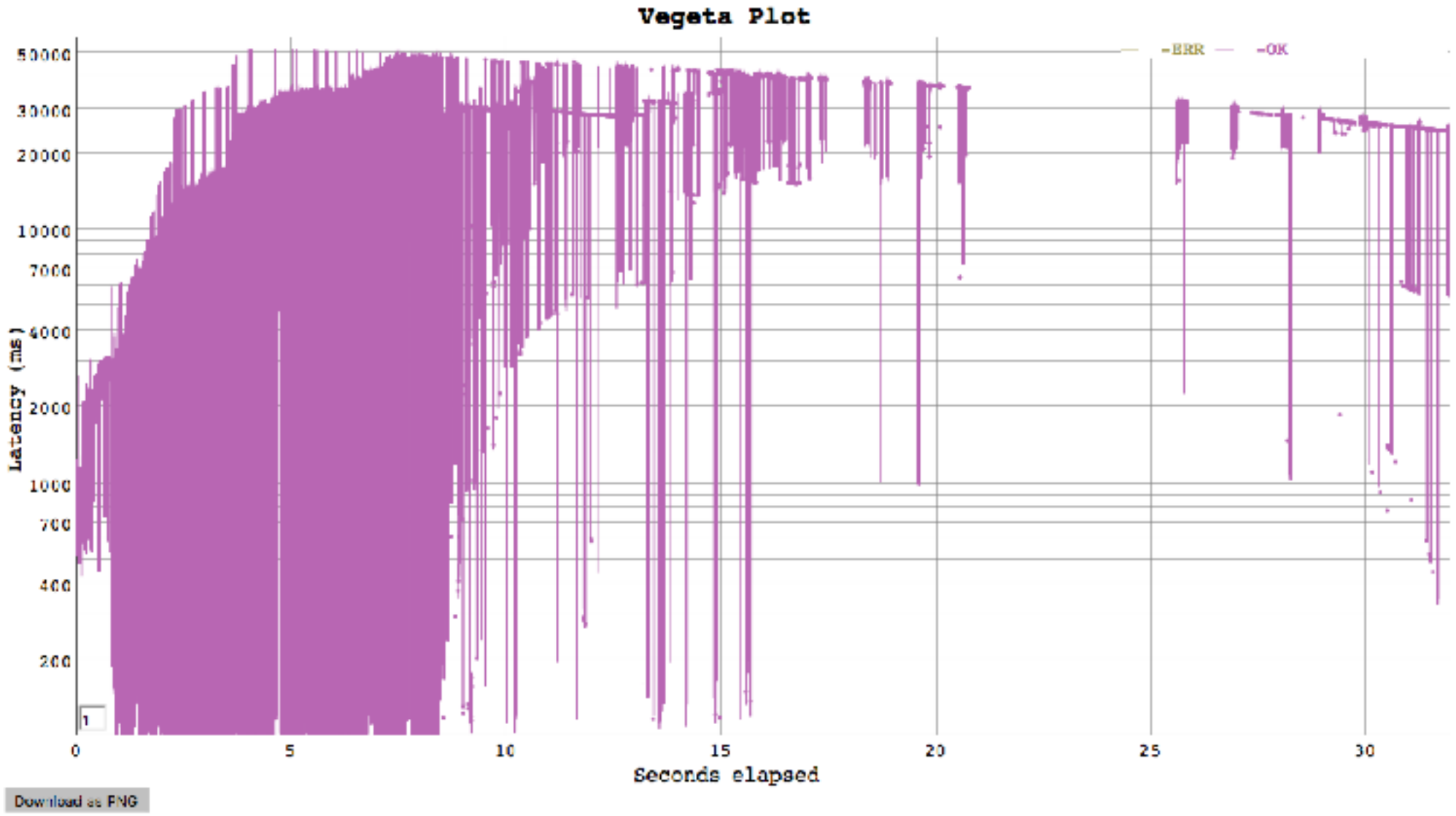
Example with existing code/libraries

- Assume servlet or REST service that spends a long time waiting

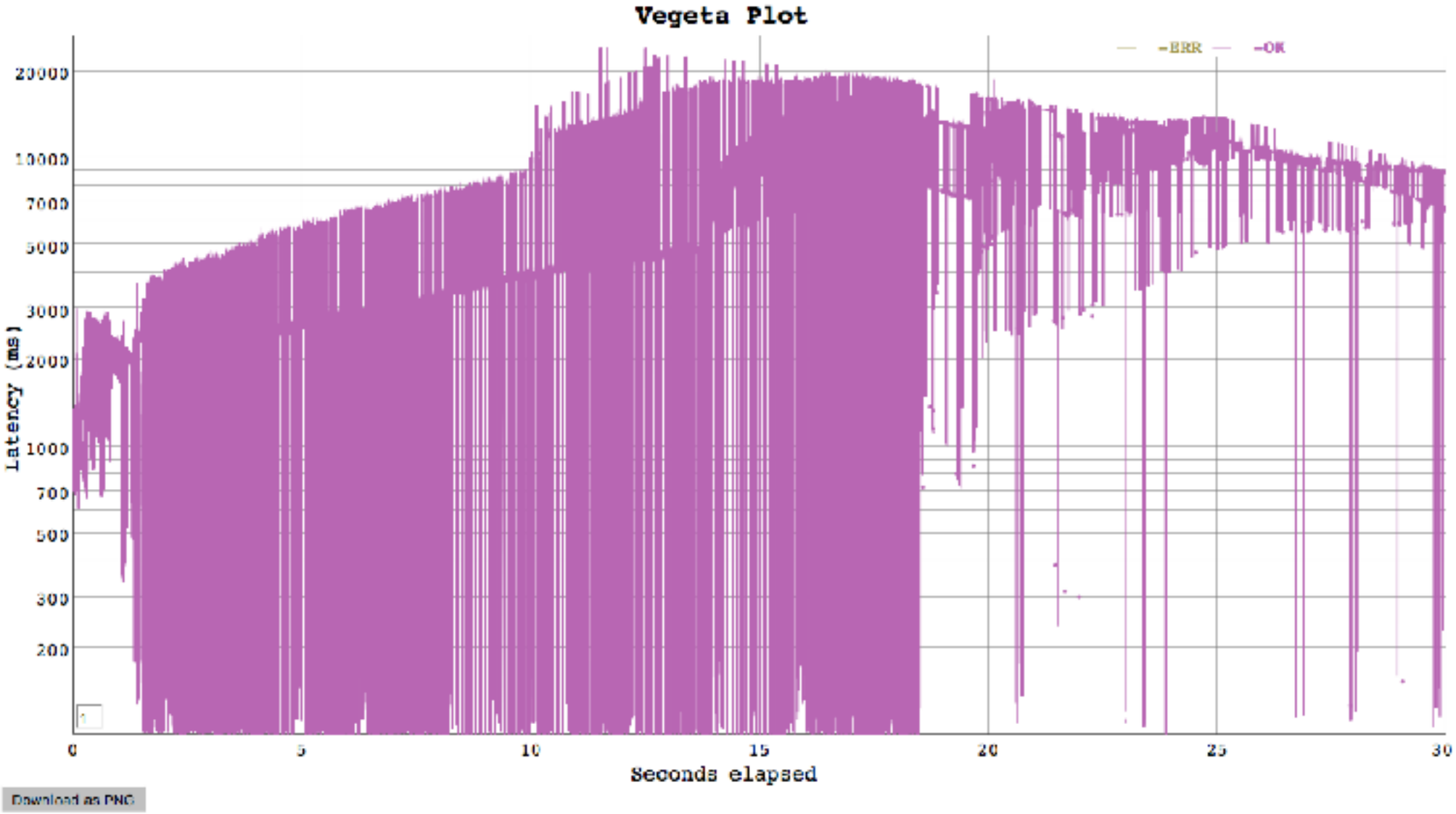
```
@GET
@Path("greeting")
@Produces(MediaType.APPLICATION_JSON)
public String greeting() {
    return "{ \"message\": \"" + computeValue() + "\" }";
}
```

assume this takes 100ms

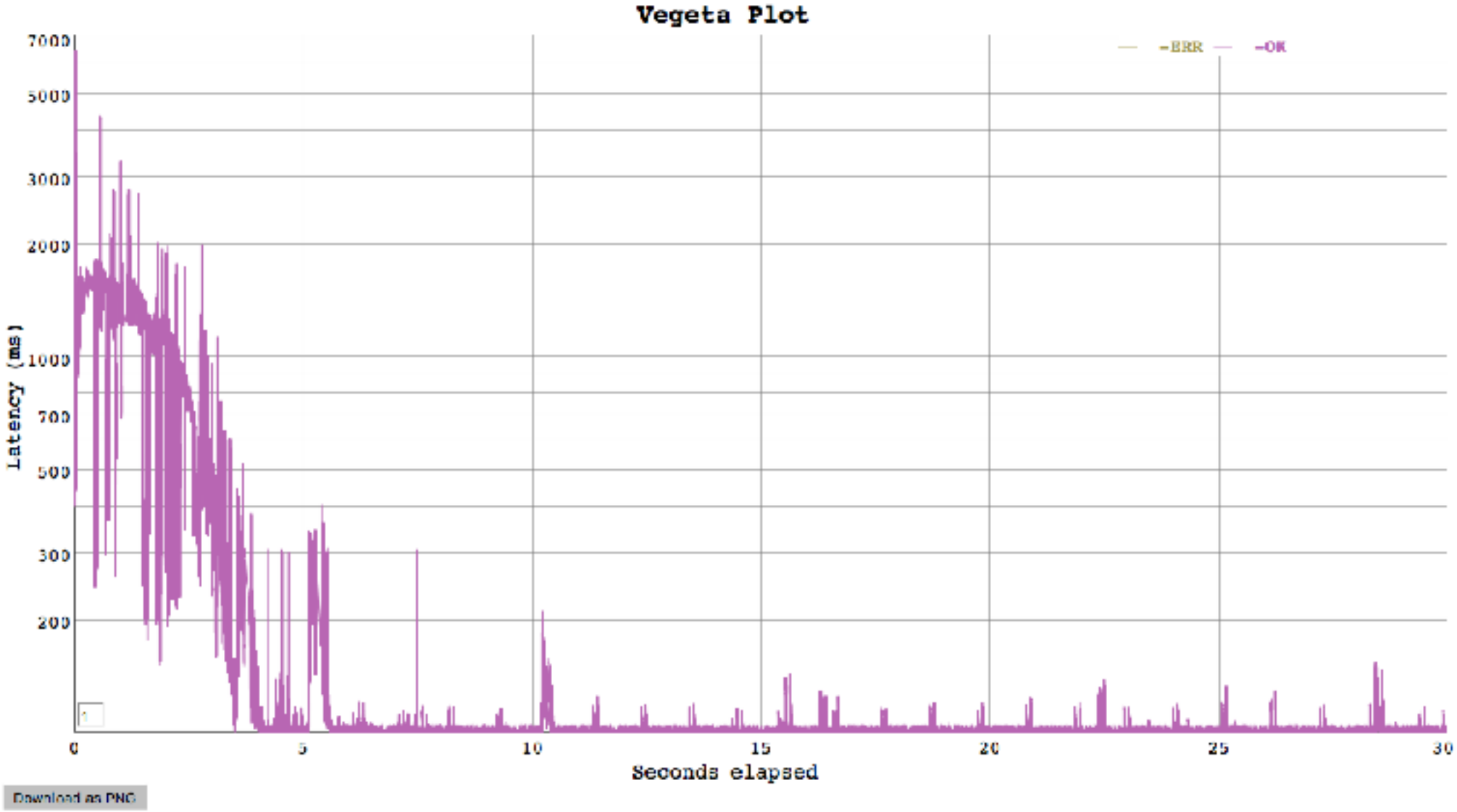
Default configuration (maxThreads = 200), load = 5000 HTTP request/s



maxThreads = 400, load = 5000 HTTP request/s



fiber per request, load = 5000 HTTP request/s



Current limitations

- Can’t yield with native frames on continuation stack
- Can’t yield while holding a monitor
- In both cases, parking pins the carrier thread
- monitorenter/Object.wait may park carrier thread

Back to the big questions

- Will fibers be able to run *all existing code*?
- Should we completely re-imagine threads?

Thread.currentThread() and Thread API

- A lot of existing code uses the Thread API and `Thread.currentThread()` (maybe indirectly)
- For now, current prototype can run in a *mode* that emulates `Thread.currentThread()` and most of the Thread API. That allows fibers to run existing code.
- Project Loom is the opportunity to re-imagine threads

What is wrong with java.lang.Thread

- ThreadGroup
- Context ClassLoader
- Inheritance: TCCL, ACC, InheritedThreadLocals
- suspend/resume, deprecated for 20+ years
- Thread interrupt problematic with threads pools
- Thread locals …

Thread locals

- e.g. container managed cache of connection or credentials context
- Long-standing source of memory leaks in thread pools
- Often used because because isn’t anything better
  - Sometimes used to make context available to callees
  - Sometimes used as approximation to “processor locals”

Locals (exploring)

- Frame/scope locals
  - Locals that are accessible to callees
    e.g. Clojure dynamic binding, special variables in Lisp
  - Semantics TDB
  - Maybe tied with S*tructured Concurrency*
- Processor locals
  - Locals keyed on cpu ID rather than Thread
  - Potential users are Striped64/LongAddr to avoid needing fields in Thread

Structured Concurrency (exploring)

- Core idea
  "*every time that control splits into multiple concurrent paths, we want to make sure that they join up again*".
- Background reading and motivations:
  - Nathaniel J Smith blogs:
    - Notes on structured concurrency, or: Go statement considered harmful
    - Timeouts and cancellation for humans
  - Also Martin Sustrik blogs on state machines and structured concurrency in high-level languages
- Implemented as *Nurseries* in Python Trio library

Structured Concurrency

- Early prototype, but not in loom/loom yet

```
Instant deadline = Instant.now().plusSeconds(1);
FiberScope.withDeadline(deadline).run(() -> {

    Fiber<?> fiber1 = …
    Fiber<?> fiber2 = …

});
```

fiber1 and fiber2 guaranteed to have terminated

Communication between fibers

- Current prototype executes tasks as Runnable or Callables
- j.u.concurrent *just works* so can share objects or share by communicating
- Not an explicit goal at this time to introduce Channels or other concurrency APIs but new APIs may emerge

Current status

- Initial prototype with Continuation and Fiber support
- Current focused on
    - Performance
    - Fiber API
    - Debugger support
- Several other topics under exploration

APIs that potentially park in current prototype

- Thread sleep, join
- java.util.concurrent and LockSupport.park
- Networking socket read/write/connect/accept
- Pipe read/write

Footprint

- Thread
  - Typically 1MB reserved for stack + 16KB of kernel data structures
  - ~2300 bytes per started Thread, includes VM meta data
- Fiber
  - Continuation stack: hundreds of bytes to KBs
  - 200-240 bytes per fiber in current prototype

Debugging and serviceability

- Basic support in JVM TI to track fiber scheduling, mount and unmount
- Hope to have some basic debugger support soon
- No investigation yet on JMX/java.lang.management and other tool APIs

Other topics to explore

- Tail calls
- Forced preemption
- Serialization and cloning

More information

- Project Loom page: http://openjdk.java.net/projects/loom/
- Mailing list: loom-dev@openjdk.java.net
- Repo: http://hg.openjdk.java/net/loom/loom (fibers branch)

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


Java™
ORACLE