STM-Based Futures and Promises in Scala

Andreas Schwarz
joint work with Martin Sulzmann

Abstract

Futures and promises are one approach to make concurrent programming easier. Programming with futures and promises feels close to the common 'sequential' programming model. The underlying implementation carries out computation steps asynchronously as much as possible by relieving the user from the burden of explicit thread management
In this talk, we review futures and promises in the Scala programming language. Scala supports futures and promises via an API and provides a wide range of combinators to compose them in sophisticated ways. We have implemented our own futures and promises API based on the concept of Software Transactional Memory (STM). We are able to provide some interesting extensions of futures and promises. Experimental results show that the STM-based API is competitive compared to the native API.

Agenda

1. Futures & Promises in Scala
2. Idea of STM-based Futures
3. Idea of MultiTrySuccess
4. Performance comparison of the implementations

What are Futures & Promises good for?

- Coordination of asynchronously executed computations
- For example
  - CPU intensive computations like matrix inversion
  - Loading data from a database
  - HTTP Requests

Futures are ...

- used to coordinate asynchronously executed computations
- placeholder that represents an unknown or uncompleted computation
- read-only
- can be queried several times

Futures in Scala

```
val future = Future {
  io.Source.fromURL("http://www.webservice1.de/").mkString
}
// returns None due calculation is still in progress
val r1 = future.value
// waits until the result is calculated and then yields it
val r2 = Await.result(future, Duration.Inf)
// using a closure which is executed after the future completes
var r3: String = null
future onSuccess { case response => r3 = response }
```

Futures are composable

```
val future = Future {
  io.Source.fromURL("http://www.webservice1.de/").getLines()
} fallbackTo { // alternative path
  Future {
    io.Source.fromURL("http://www.webservice2.de/").getLines()
  }
} map { // use map to alter the data
  lines => lines.mkString
} recover { // recover in case something went wrong
  case e: Exception => "Some error message"
}
```

Futures in Scala

- Pros
  - API-based
  - Has an expressive set of combinators
  - Part of the Scala Standard Library
  - Optimized
- Cons
  - Complex implementation (6 classes and traits)
  - Extensions complicated due several classes must be altered

Idea of STM-based Futures

- STM-Based Futures API
- STM fits pretty well to Future semantics
- Usage of the functional programming paradigm

STM in Scala

- Atomic memory operations
- “All or nothing” semantics (ACID)
- User can invoke rollback (retry)
- composable

STM in Scala II

```
val queue = Ref(Queue.empty[Int])

def put(value: Int): Unit = {
  atomic {
    queue enqueue value
  }
}

def pop (): Int = {
  atomic {
    if ( queue isEmpty ) retry
    queue dequeue
  }
}
```

Futures via STM

- Future represented as a transactional state variable:
  - Empty
  - Success
  - Failure

Creating a STM-Based Future

```
final case class Future[A]() {
  val callbacks = new ConcurrentLinkedQueue[Runnable]()
  val result: Ref[Result[A]] = Ref(Result.empty)
}

object Future {
  def apply[A](func: => A): Future[A] = {
    val future = new Future[A]
    execute(createRunnable(complete(future, func)))
    future
  }
}

private def complete[A](future: Future[A], value: A) = {
  atomic { implicit tnx => future.result() = Result(value) }
  afterCompletion(future)
}
```

STM-Based Blocking Await (Get)

```
def get[A](future: Future[A]): Option[A] =
  atomic { implicit txn =>
    future.result() match {
      case Empty => retry
      case Fail => return None
      case Success(x) => return Some(x)
    }
  }
```

STM-Based OnSuccess

```
def onSuccess[A](future: Future[A], callback: (A) => Unit) = {
  val cb = createRunnable {
    get(future) match {
      case None => ()
      case Some(x) => callback(x)
    }
  }
  addOrExecuteCallback(future, cb)
}
```

STM-Based Future API

- Support of all common combinators
- No exception-handling. Failure represented as a data type
- “Callback” trick
- Some extensions are possible now!

STM-Based Promises

```
final class Promise[A] {
  val future: Future[A] = new Future[A]
}

object Promise {
  def apply[A]() = new Promise[A]
}

def trySuccess[A](promise: Promise[A], result: A): Boolean = {
  atomic { implicit txn =>
    promise.future.result() match {
      case Empty => complete(promise.future, result)
        true
      case _ => false
    }
  }
}
```

Dining Philosopher Example

```
def philosopherRace(rounds: Int): Unit = {
  for (a <- 0 until rounds) {
    val forks = // create promises that represent forks
    val philosophers = // create list of philosophers
    philosophers foreach { philo =>
      onSuccess(philo, (p: Philosopher) => println(p.id))
    }
  }
}

def philosopher(id: Int, left: Promise[Int],
                         right: Promise[Int]) = {
  Future {
    val succ1 = trySuccess(left, id)
    val succ2 = trySuccess(right, id)
    (succ1, succ2) match {
      case (true, true) => Philosopher(id)
      case _ => null
    }
  }
}
```

STM-Based MultiTrySuccess

- Simple extension to complete several Promises atomically
  - If all Promises are empty, they are completed
  - Otherwise, we are not doing anything
- Promises are STM-Based!
- Above is easy to implement

MultiTrySuccess

```
def multiTrySuccess[A](ps: List[(Promise[A], A)]): Boolean = {
  atomic { implicit txn =>
    val allEmpty = ps.forall(p => p._1.future.result() == Empty)
    allEmpty match {
      case true =>
        ps.forall(p => trySuccess(p._1, p._2))
      case _ => false
    }
  }
}
```

Why don't we extend Scala's Promises?

- Requires the extension of several classes
- Locks or another concurrency mechanism are necessary
- Requires some order function to lock and unlock in proper order
- Such a function itself is nontrivial to implement

Performance comparison

- Evaluation and comparison of performance necessary
- Usage of ScalaMeter 0.7
  - Benchmarking Framework
  - Built-in warm up phase
  - Automatic outlier detection
  - Runs in separate JVM
- Tests are executed on a server with 64 cores

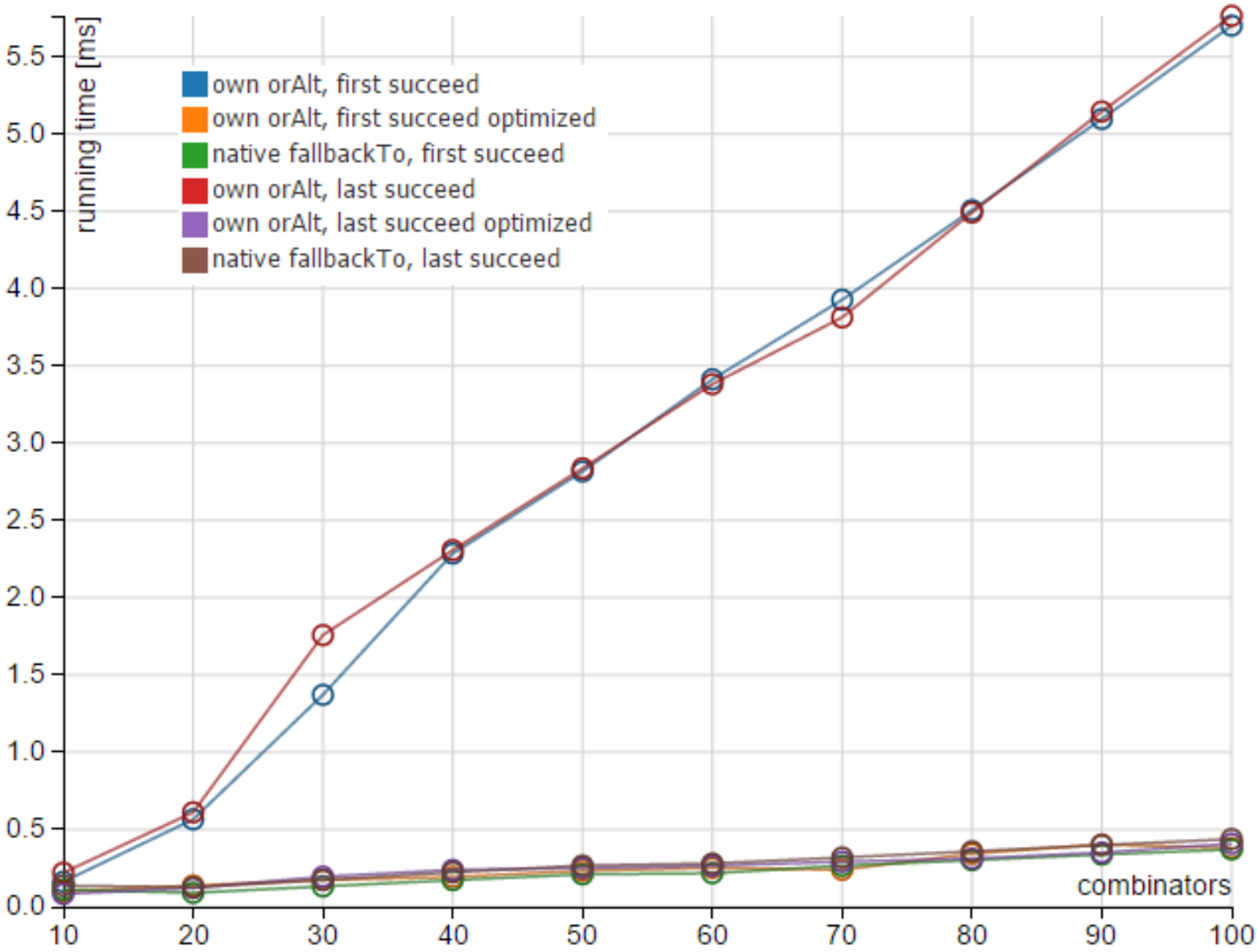
Calculation of alternatives I

- Nesting of several alternative calculations
- Two use cases
  - The first calculation succeeds all others fail
  - The last calculation succeeds all others fail

Calculation of alternatives II

```
object Benchmark extends Bench.OfflineReport {
  // how data will be aggregated
  override def aggregator = Aggregator.median
  // dsl to generate input data
  val combinations = Gen.range("combinators")(10, 100, 10)
  // the logic of benchmark
  performance of "Calculation of alternatives" in {
    measure method "orAlt, first succeed" in {
      using(combinations) in { toCombine =>
        var f = future.Future(42)
        for (i <- 1 to toCombine) {
          f = future.orAlt(f, future.fail())
        }
        future.get(f)
      }
    }
  }
}
```

Calculation of alternatives III



Calculation of alternatives IV

- first approach by far slower than the others
- first approach scales really bad
- optimized version as fast as the native ones
- nearly no difference between the use cases

Conclusion

- STM fits perfectly to this abstraction
- No significant performance differences of the STM-based API
- Extensions of the abstraction possible
- Source code available:
  - https://github.com/anschwar/FutureSTM

Appendix

further details, benchmarks

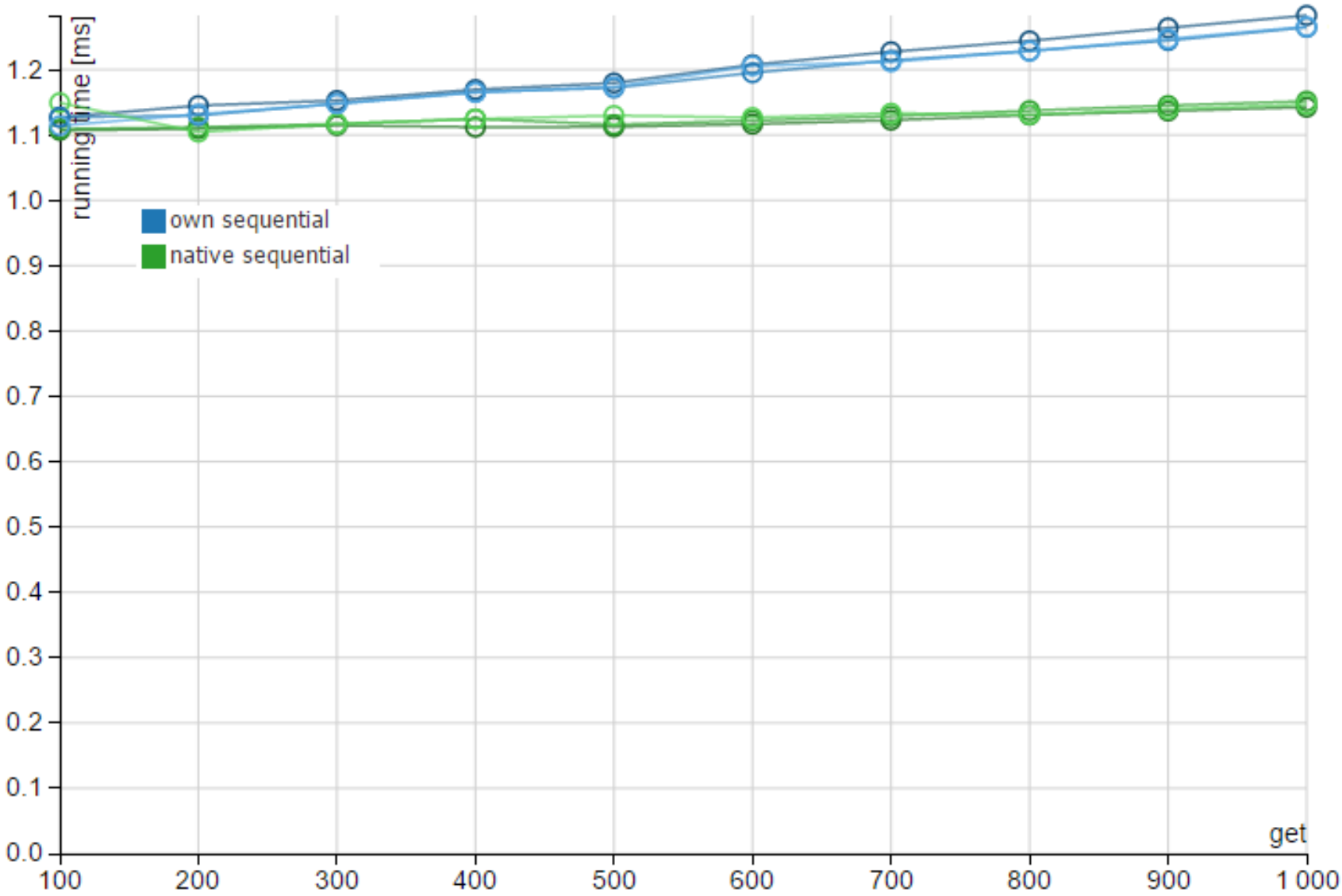
Futures & Promises in Scala

- Futures & Promises are partial implemented Traits
- Object creation via apply function
- Usage of methods on the object directly
- Represented by the same object (DefaultPromise)
- Offers no method to wait for the ending of the calculation
- Highly optimized
  - Centralized execution of tasks
  - Tasks will be combined if possible

Sequential reading I

- One Future will be completed
- Calculation is delayed 1 ms
- Read the result 100-1000 times

Sequential reading II



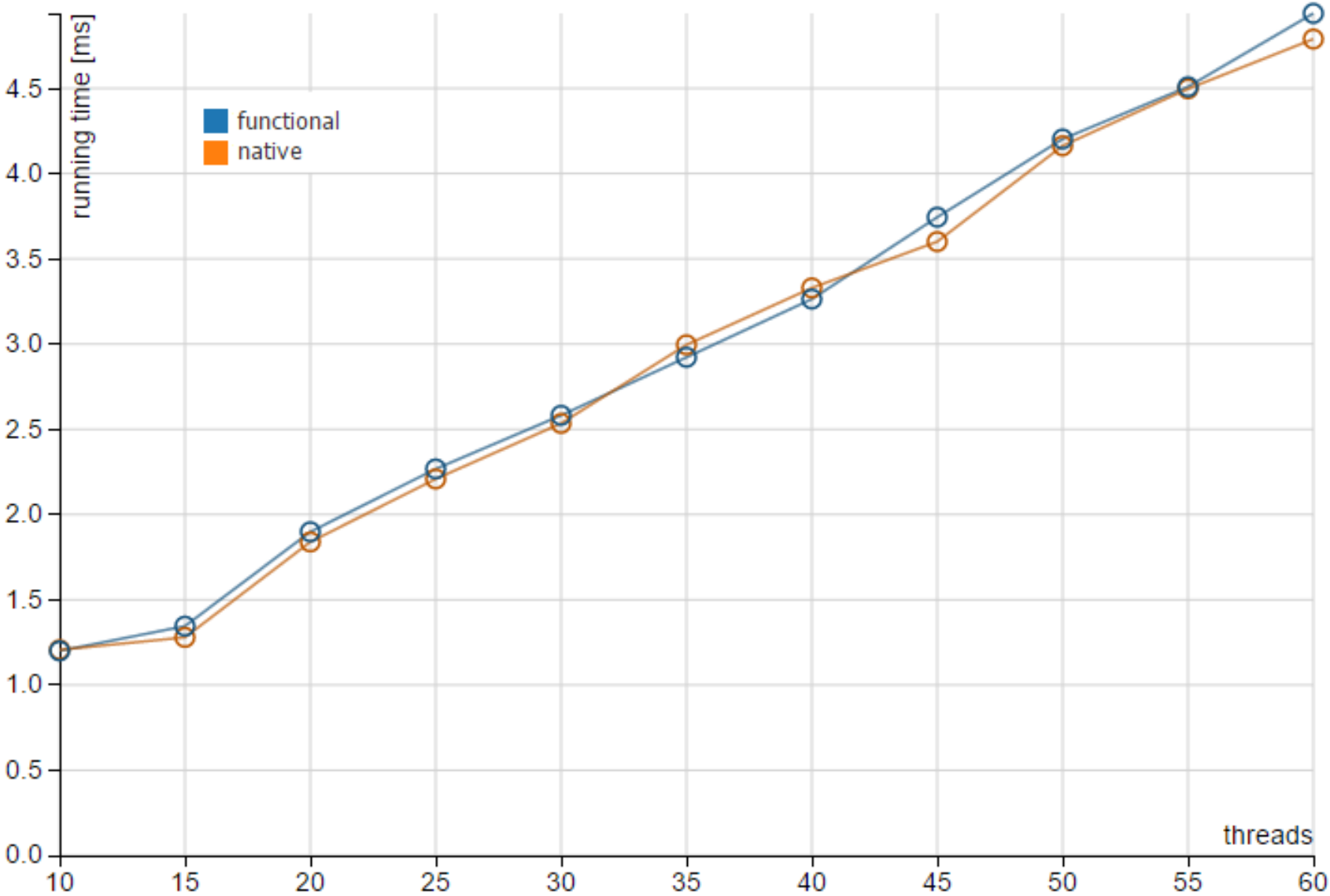
Sequential reading III

- Native implementation is up to $\sim$ 0.2 ms faster
- Native implementation is nearly constant
- Own API has linear performance
- Worse performance is caused by the overhead of STM

Concurrent reading with Threads I

- One Future will be completed
- Calculation is delayed 1 ms
- 10 to 60 Threads read the result concurrently
- Each thread is created with the new operator

Concurrent reading with Threads II



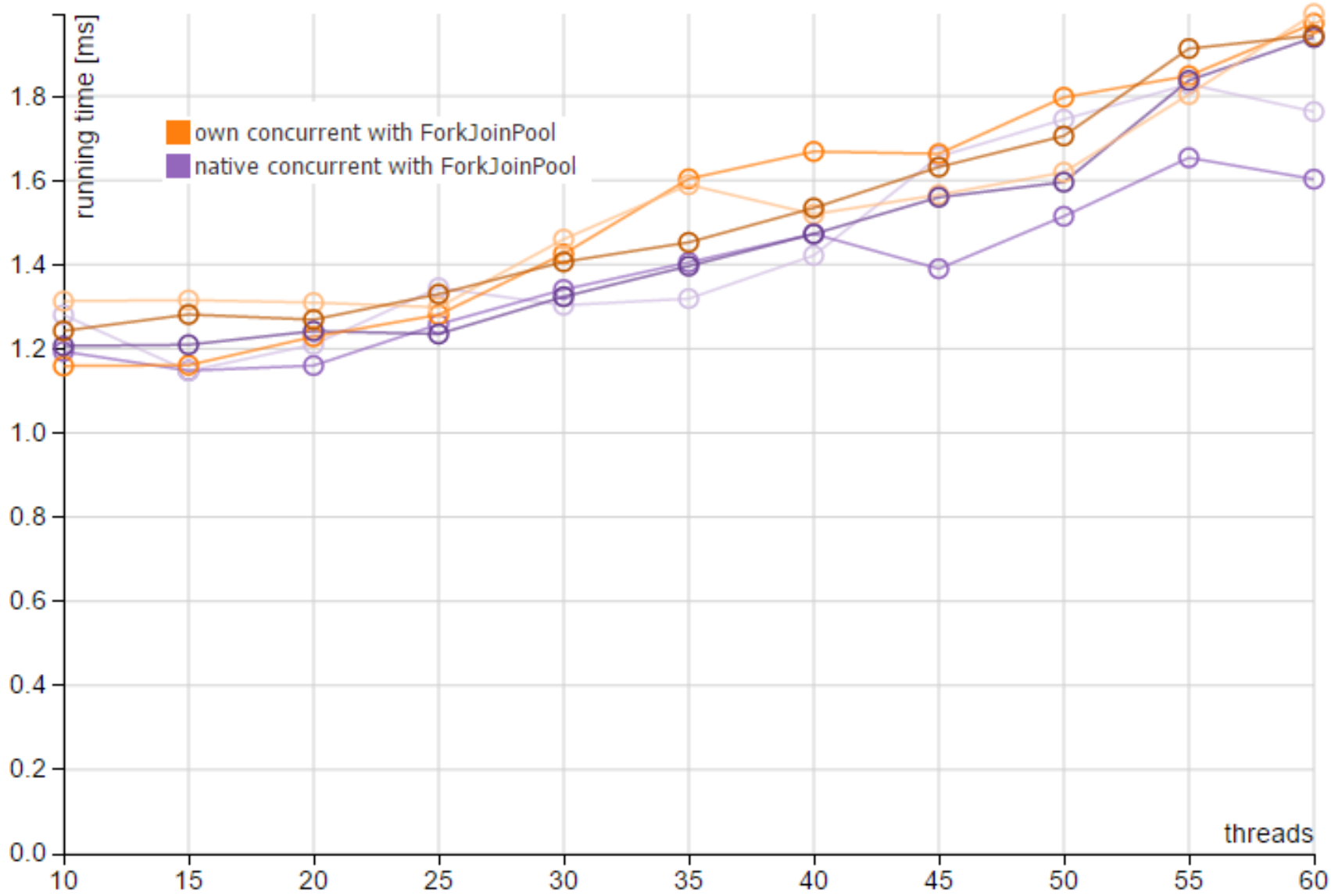
Concurrent reading with Threads III

- Both have the same performance characteristics
- Runtime analysis show that the creation of threads consumes much time

Concurrent reading with a Threadpool I

- Again one Future will be completed
- Calculation is delayed 1 ms
- 10 to 60 Threads read the result concurrently
- Usage of a Threadpool to reuse existing threads

Concurrent reading with a Threadpool II



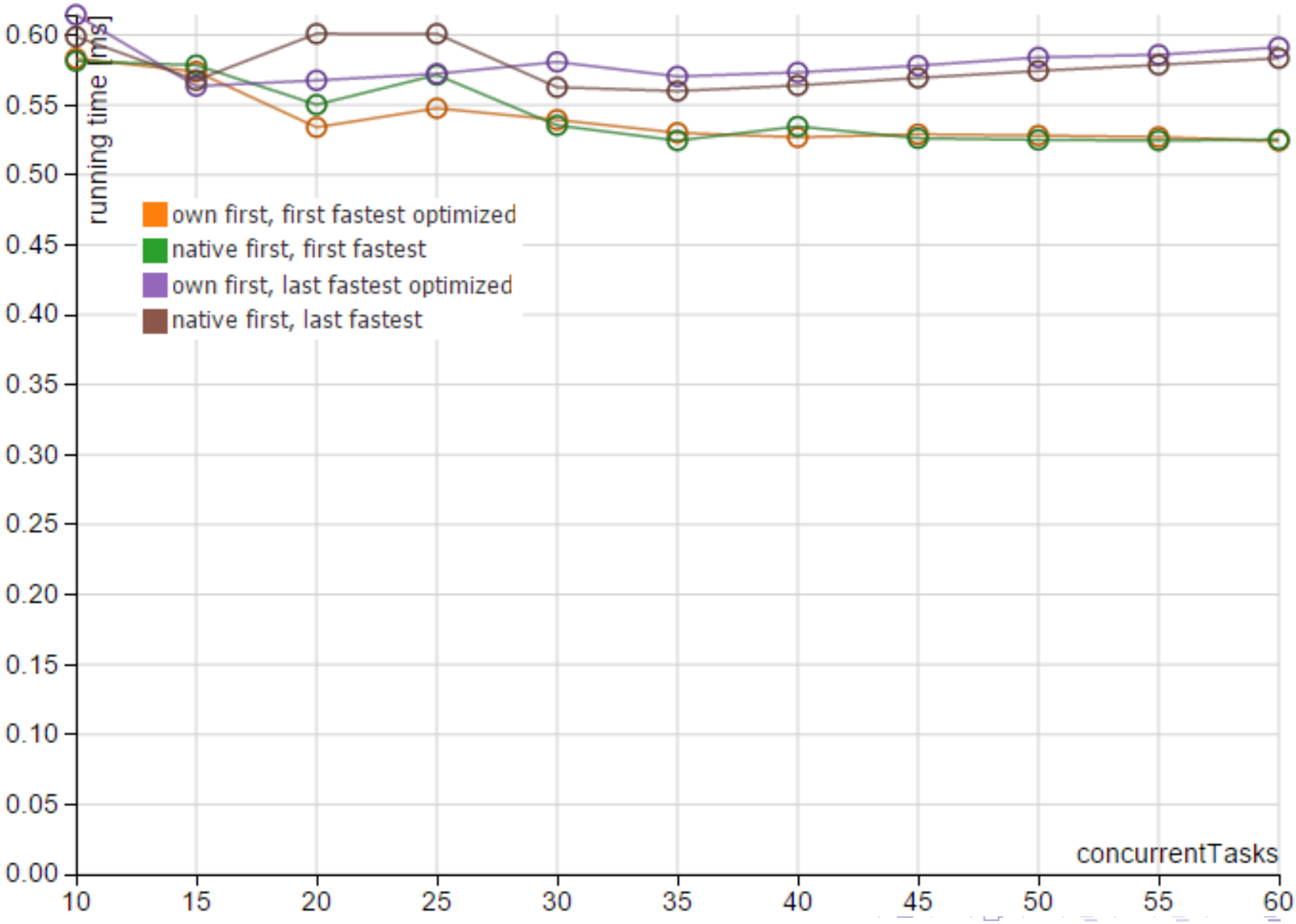
Concurrent reading with a Threadpool I

- Calculation more than 2 times faster
- Partially strong variations cause the the threading
- Native implementation slightly faster due the overhead of STM

First finished Future I

- First function is used to determine the first completed future
- Usage of busy wait due Thread.sleep() is not fine grained enough
- Calculation delayed 500 + x microseconds
- x depends on the position of the future (first 0, last 60)
- Two use cases
  - The first Future is the fastest
  - The last Future is the fastest

First finished Future II



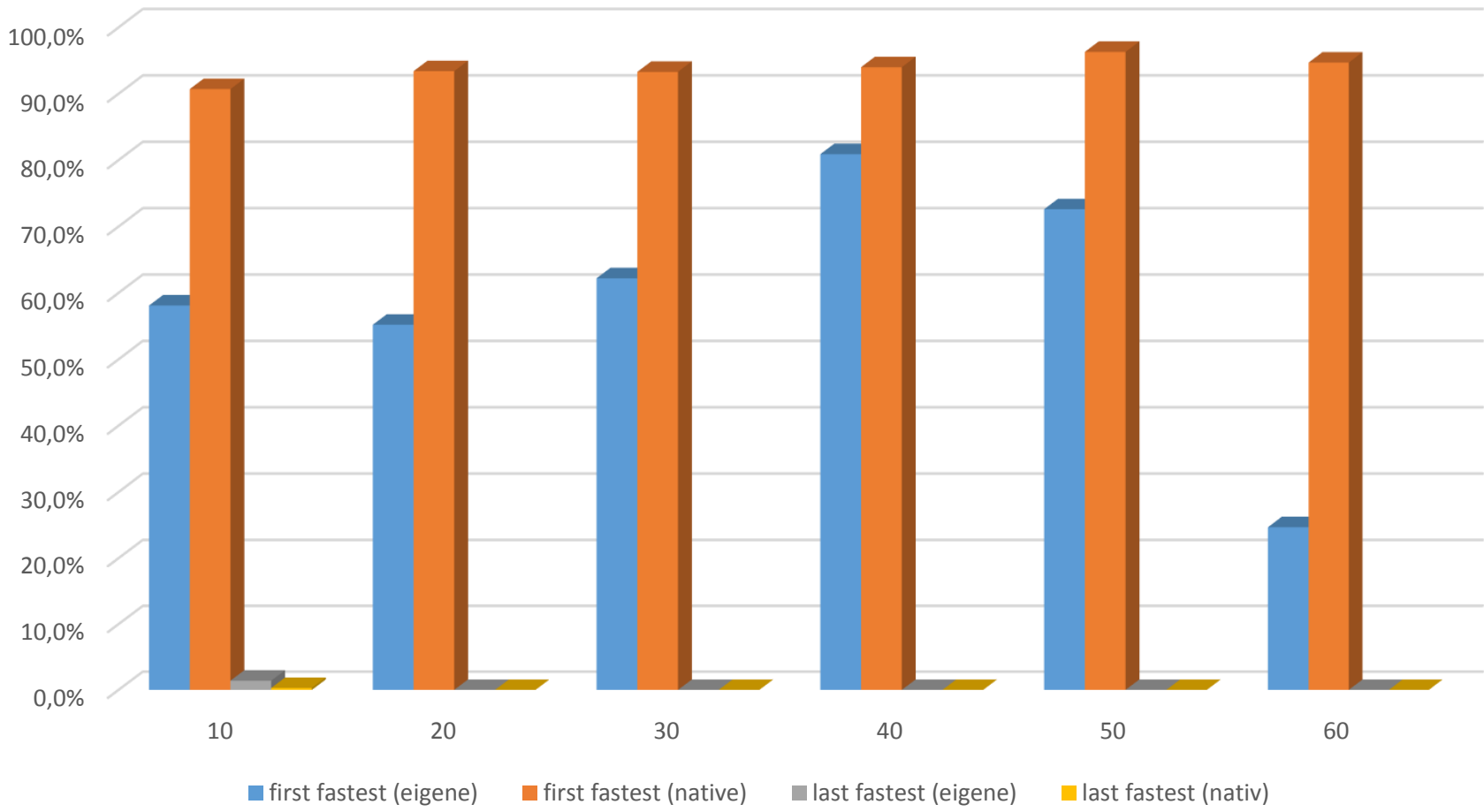
First finished Future III

- Strong variations at the beginning
- Normalizes at about 30 Futures
- Performance at above 30 Futures is nearly the same

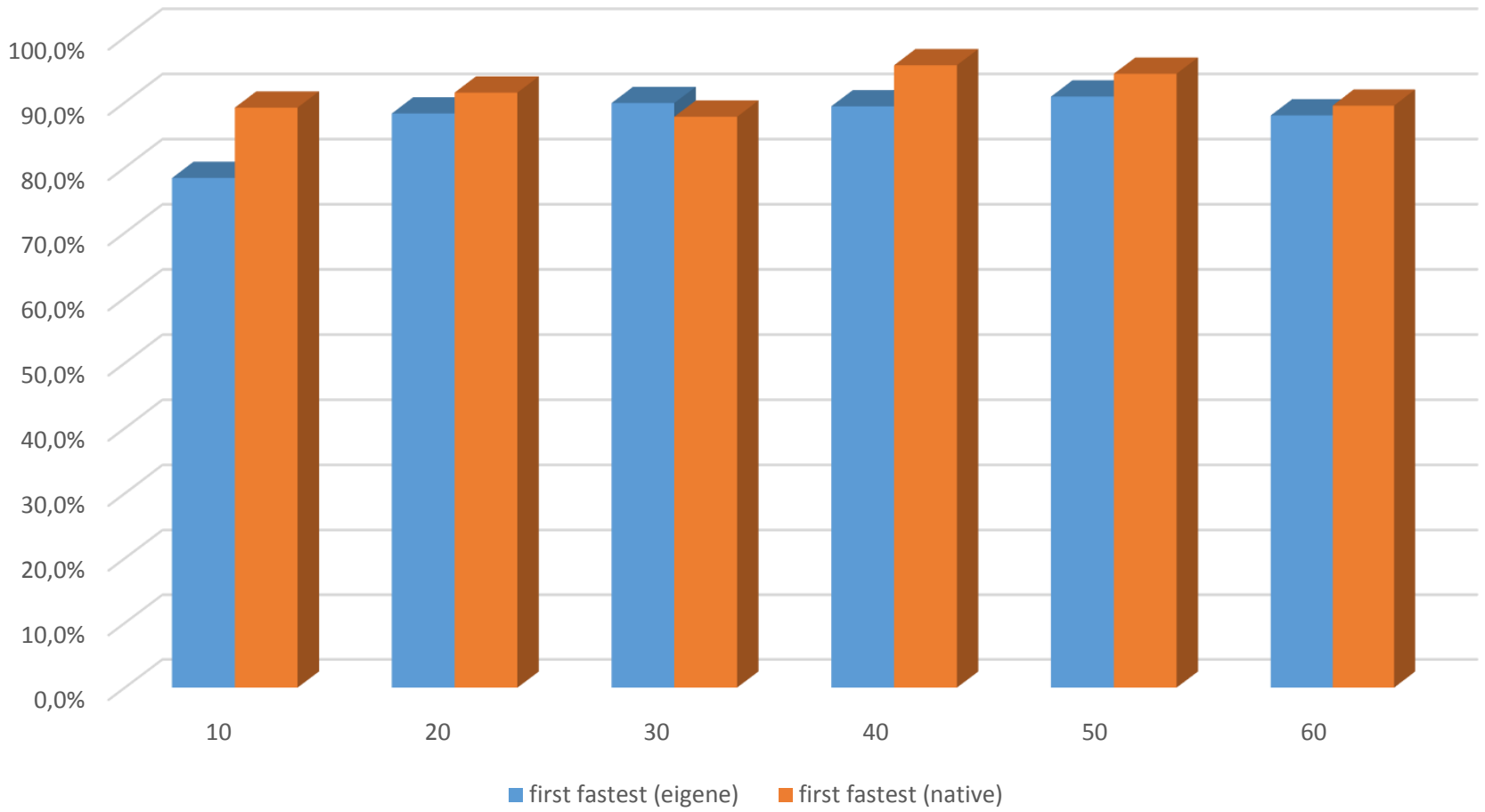
First finished Future (accuracy) I

- Compare the accuracy of the function
- Determine how often the first Future completes first
- First compare the not optimized version with the native
- Second compare the optimized version with the native

First finished Future (accuracy) II



First finished Future (accuracy) III



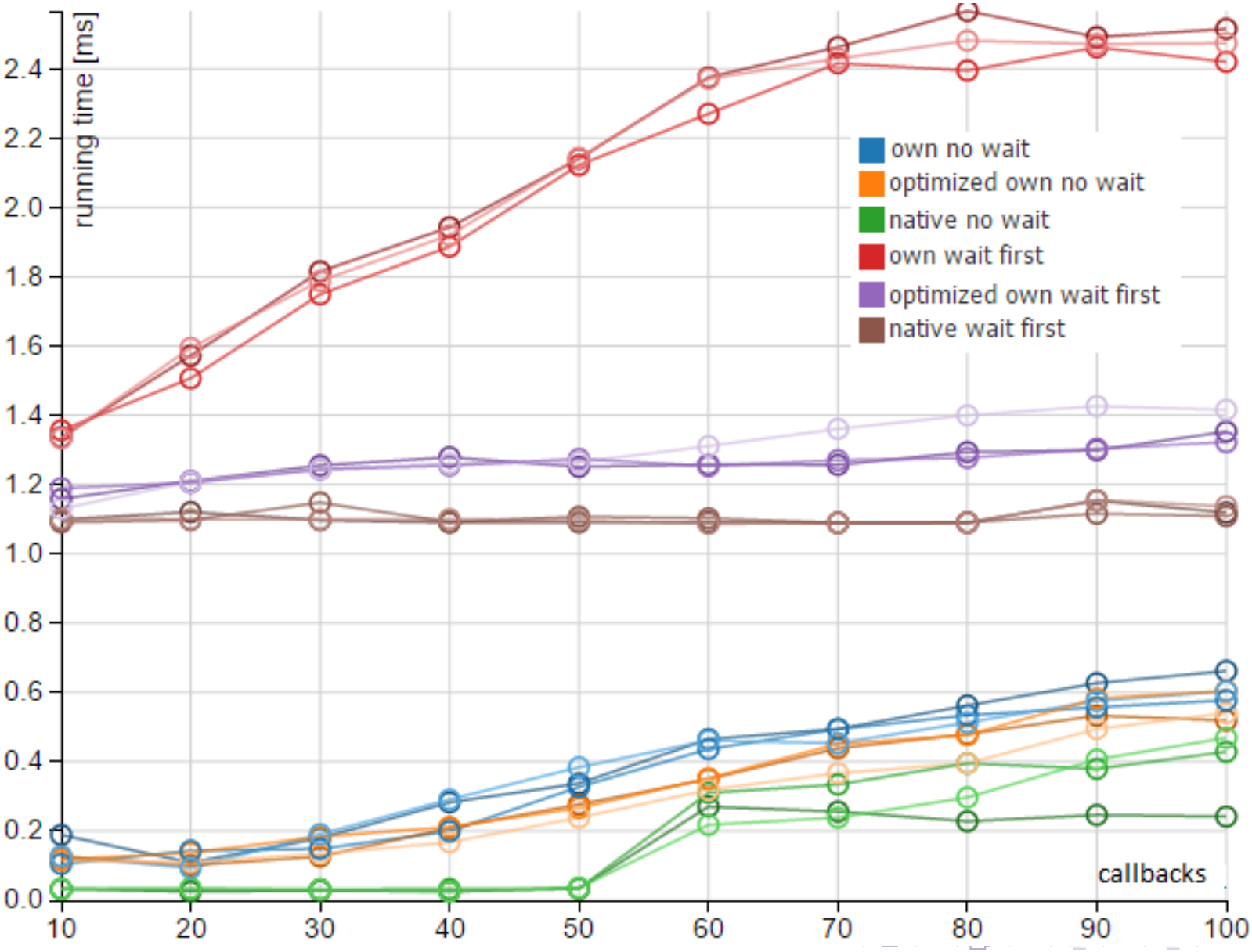
First finished Future (accuracy) IV

- Not optimized version has a bad accuracy especially with much Futures
- Second use case is not meaningful
- Accuracy of the native Version is nearly always better
- But optimized version is nearly equal

Appending of multiple callbacks I

- several tasks will be added as callback
- they will be executed after the future completes
- Two use cases
  - Future completes immediately
  - Future completes after a 1 ms delay

Appending of multiple callbacks II



Appending of multiple callbacks III

- Native is again the fastest
- Optimized Version has equivalent performance characteristics if the calculation is delayed