Reactive Async: Expressive Deterministic Concurrency

Philipp Haller Simon Geries
KTH Royal Institute of Technology, Sweden
phaller@kth.se

Michael Eichberg Guido Salvaneschi
TU Darmstadt, Germany
{eichberg, salvaneschi}@cs.tu-darmstadt.de

Abstract

Concurrent programming is infamous for its difficulty. An important source of difficulty is non-determinism, stemming from unpredictable interleavings of concurrent activities. Futures and promises are widely-used abstractions that help designing deterministic concurrent programs, although this property cannot be guaranteed statically in mainstream programming languages. Deterministic-by-construction concurrent programming models avoid this issue, but they typically restrict expressiveness in important ways.

This paper introduces a concurrent programming model, Reactive Async, which decouples concurrent computations using so-called *cells*, shared locations which generalize futures as well as recent deterministic abstractions such as LVars. Compared to previously proposed programming models Reactive Async provides (a) a fallback mechanism for the case where no computation ever computes the value of a given cell, and (b) explicit and optimized handling of cyclic dependencies. We present a complete implementation of the Reactive Async programming model as a library in Scala. Finally, the paper reports on a case study applying Reactive Async to static analyses of JVM bytecode based on the Opal framework.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – Concurrent, distributed, and parallel languages

Keywords asynchronous programming, concurrent programming, deterministic concurrency, static analysis, Scala

1. Introduction

Developing correct concurrent programs is a well-known challenge. Concurrency can create race conditions that may corrupt the state of the program. Also, concurrent programs are hard to reason about, because multiple flows of execution can generate non-deterministic results because of their unpredictable scheduling. Traditionally, developers address these problems by protecting state from concurrent access via synchronisation. Yet, concurrent programming remains an art: insufficient synchronisation leads to unsound programs but synchronising too much does not exploit hardware capabilities effectively for parallel execution.

Over the years researchers have proposed concurrency models that attempt to overcome these issues. For example the actor model [13] encapsulates state into actors which communicate via asynchronous messages–a solution that avoids shared state and hence (low-level) race conditions. Yet, the outcome of the program is potentially non-deterministic because it depends on how actors execution, message sending and message receiving interleave. Such issues are shared by a number of popular concurrency models, including STM [12] and futures/promises [17].

Recently, researchers designed concurrency models, like LVars [15], FlowPools [23] and Isolation Types [6], that result in deterministic executions. The idea behind these models is to restrict expressivity to capture only computations that can be defined in a deterministic way.

However, the limitations imposed by the deterministic concurrency models proposed so far are severe. They push to the programmer the burden of merging the result of concurrent computations, they do not support cyclic dependencies–which are common in a number of fixed point algorithms, *e.g.,* in static analysis–and they do not support dynamic dependencies among concurrent computations. Adding and removing dependencies dynamically is critical for certain applications where concurrent abstractions are created on a very large scale and must be removed when they are no longer required.

In this paper, we present Reactive Async, a programming model that allows developers to write deterministic concurrent code in a composable way, supports *dynamic cyclic dependencies* and supports a *fallback mechanism* to resolve computations that haven't reached final values nor have pending dependencies,

Our evaluation shows that Reactive Async scales well on multicore architectures, is efficient and can express complex computations that are not supported by existing deterministic concurrency models in real-world scenarios.

```
type ConcurrentSet[T] = TrieMap[T, Int]

def traverse(v: Vertex,
             userIDs: ConcurrentSet[Int]): Unit = {
  if (isInteresting(v))
    userIDs.put(v.userID, 0)
  for {n <- v.neighbors; if (!visited(n))} {
    Future {
      traverse(n, userIDs)
    }
  }
}
```

Figure 1. Concurrent traversal and collection of IDs

```
val selectedUsers: List[Int] = ...
val entry: Vertex = ...
val userIDs: ConcurrentSet[Int] =
  TrieMap.empty[Int, Int]
// traverse graph starting from ‘entry‘
// collect users in ‘userIDs‘
traverse(entry, userIDs)
// analyze selected users
for (user <- selectedUsers) {
  val isInteresting = userIDs.contains(user)
  ...
}
```

Figure 2. Traversal, collection, and analysis of social graph

In summary, the contributions of this paper are:

- We present *Reactive Async*, a deterministic concurrency model that supports cyclic dependencies, dynamic dependencies, and fallbacks.
- We identify patterns from real-world large-scale use cases where existing approaches fall short. We show that *Reactive Async* can capture such complex computations significantly more concisely than the original implementation.
- We provide an efficient implementation of *Reactive Async* in Scala[1] that in our benchmarks exhibits scalability comparable to the state-of-the-art Scala futures [11]. In real-world case studies, the performance of *Reactive Async* achieves a speedup between 1.75x and 10x and scalability over multicore architectures is improved.

The paper is structured as follows. Section 2 provides an overview of our programming model. Section 3 describes relevant implementation details. Section 4 presents our case studies. Section 5 provides a performance evaluation. Section 6 reviews related work, and Section 7 concludes.

2. Overview

In this section, we introduce an example of concurrent programming, show the issues that arise using traditional solutions and explain how Reactive Async can address them. Next, we present in detail the features of Reactive Async.

2.1 Motivating Example

Consider the example of a social network modeled as a graph where vertices represent users of the social network and edges represent connections between users (*e.g.,* a "follows" relation). Suppose an application traverses a given social graph and collects the IDs of users satisfying a predicate ("interesting" users) in a set. In order to leverage multicores, the traversal and collection should be performed concurrently. Therefore, user IDs should be collected in a thread-safe (concurrent) set. Figure 1 shows a simple concurrent breadth-first graph traversal for such a purpose. Note that we use a concurrent `TrieMap` [22] as the implementation of a concurrent set where elements of the set are represented as keys of the `TrieMap`. Now, suppose that in addition to collecting interesting users, we would also like to analyze selected users in more detail as shown in Figure 2. This approach suffers from two major issues:

1. Since the graph traversal is concurrent (spawning futures for the traversal of neighbors), the invocation of `contains` on line 11 returns a non-deterministic result. A selected user may be "interesting", but the concurrent traversal may – at the point when `contains` is invoked – not have added the corresponding ID to the `userIDs` set.
2. Furthermore, without examining the implementation details of the `traverse` function, it is not clear whether the concurrent traversal yields a deterministic outcome in the first place (*e.g.,* if user IDs are not only added to the `userIDs` set but may also be removed in certain cases).

These issues demand for a programming model that leads to a deterministic result still preserving the efficiency of concurrent computation.

2.2 Reactive Async

Using Reactive Async, both issues can be addressed in a straightforward way. First, instead of using a concurrent set, user IDs are maintained in a so-called *cell*. Computations performed on cells are made deterministic by requiring cells to contain values taken from a *lattice*. This enables *concurrent, monotonic updates* of the value of a cell while preserving the *determinism* of the (concurrent) computation. Cell updates are performed using *cell completers*.

Figure 3 shows the code of the `traverse` function, but this time using a cell completer for updating `userIDs`. In order for this cell completer to be valid, we must guarantee that the elements of type `Set[Int]` form a *lattice*. This is done by providing the (implicit) instance `IntSetLattice` of the `Lattice` type class for type `Set[Int]` (lines 1–6). Line 11 shows a monotonic update with `Set(v.userID)` which updates the value of `userIDs` with the *join* of the cell's current value and `Set(v.userID)`, effectively adding `v.userID` to the set of collected users. Importantly, during

[1] See https://github.com/phaller/reactive-async.

```scala
implicit object IntSetLattice
               extends Lattice[Set[Int]] {
  val empty = Set()
  def join(left: Set[Int], right: Set[Int]) =
    left ++ right
}

def traverse(v: Vertex,
             userIDs: CellCompleter[Set[Int]]): Unit = {
  if (isInteresting(v))
    userIDs.putNext(Set(v.userID))
  for {n <- v.neighbors; if (!visited(n))} {
      Future {
        traverse(n, userIDs)
      }
  }
}
```

Figure 3. Concurrent traversal and collection of IDs using a cell completer

```scala
val selectedUsers: List[Int] = ...
val entry: Vertex = ...
val pool = new HandlerPool
val userIDs = CellCompleter[Set[Int]](pool)
// traverse graph starting from ‘entry‘
// collect users in ‘userIDs‘
traverse(entry, userIDs)
pool.onQuiescent {
  // analyze selected users
  for (user <- selectedUsers) {
    val isInteresting = userIDs.contains(user)
    ...
  }
}
```

Figure 4. Traversal, collection, and analysis of social graph using quiescence

a concurrent traversal, multiple monotonic updates (like the one on line 11) would be applied concurrently. However, the final value of the `userIDs` cell would be unaffected by the order of these updates.

Finally, in order to safely perform further analyses of interesting users in a deterministic way (effectively avoiding a race condition), Reactive Async supports the concept of *quiescence*. Essentially, a concurrent computation on cells reaches quiescence whenever it is guaranteed that none of the cells are updated anymore. Therefore, the values of cells are stable as soon as quiescence is reached, and read accesses return deterministic results. To revisit the example, Figure 4 shows how to make the previously non-deterministic analysis deterministic. Here, the analysis is only performed when the traversal has reached quiescence. This is done by registering an *onQuiescent* handler on line 8. Therefore, the result returned by the invocation of `contains` on line 11 is now deterministic, since the `userIDs` cell is guaranteed not to be updated any more by the traversal.

Note that it would always be possible, of course, to utilize other concurrency abstractions in Scala to ensure the termination of the graph traversal (such as futures [11]). However, in this case the programmer would be on her own to ensure that the concurrent program is deterministic (or at least data-race free). The goal of Reactive Async is to provide a single programming model that integrates mechanisms such as quiescence in a way that allows writing concurrent programs that are deterministic by construction. In the following we dive into more details of the programming model.

Core Abstractions As mentioned earlier, the programming model of Reactive Async is based on two core abstractions: *cells* and *cell completers*. The two abstractions are related in a way similar to the way Scala's futures and promises [11] are related. To recall, a Scala future of type `Future[V]` may asynchronously be completed successfully with a value of type `V`; or it may asynchronously be "failed" with an exception. A Scala promise of type `Promise[V]` allows completing its associated future *at most once*. An attempt to complete an already completed promise throws an exception.

Similar to a future of type `Future[V]`, a cell of type `Cell[K, V]` may asynchronously be completed with a value of type `V` – the `K` type parameter is explained below. `Cell[K, V]` provides a *read-only* interface to access its value(s). Similar to a promise of type `Promise[V]`, a cell completer of type `CellCompleter[K, V]` allows completing its associated cell. There are two main differences between futures/promises and cells:

- Cell completers allow multiple *monotonic writes*. The written values must be elements of a lattice. In contrast, promises allow at most one write.
- Cells allow resolving *cyclic dependencies*. In contrast, dependencies between futures must be acyclic in order to avoid deadlocks.

The distinction between cell completers is motivated by the same software engineering considerations behind the futures/promises model. With this distinction, libraries can manipulate cell completers hiding from the client state updates when the concurrent computation is in progress. In contrast, the client of the library receives a cell that can only be used to access the final result as soon as it is available.

2.3 Declarative Dependencies

Reactive Async supports the declaration of dependencies between cells, such that a monotonic update of one cell triggers a monotonic update of another cell. In addition, triggered updates are conditional on the update of the original cell. For a simple example, consider two cells c_1 and c_2 holding integer values taken from the lattice of integers where the join operator computes the maximum of the two given integers (*i.e.,* `join(x, y) = max(x, y)`). Suppose we would like to express the fact that whenever c_1 is (monotonically) updated with a value v which is a multiple of 10, c_2 should be updated with v; else, c_2 should not be updated. This means that if c_1 is updated sequentially with val-

ues $1, 5, 10, 11, 21, 30, 35, 40$ then c_2 is updated with values $10, 30, 40$. This dependency can be expressed declaratively as follows:

```
c2.whenNext(c1,
  x => if (x % 10 == 0) WhenNext else FalsePred,
  None)
```

The above snippet registers a dependency of c_2 on c_1. Whenever c_1 receives an update with value v (via c_1`.putNext(v)`), it first applies the function which is the second argument of `whenNext` to v to decide whether c_2 should be updated as well, and, if so, how; the function has type `Int => WhenNextPredicate`.

The `WhenNextPredicate` type is defined as follows:

```
sealed trait WhenNextPredicate
case object WhenNext extends WhenNextPredicate
case object WhenNextComplete extends WhenNextPredicate
case object FalsePred extends WhenNextPredicate
```

A `WhenNext` result means that the dependent cell should receive an update (with the semantics of `putNext`); the update value is either the value received by the upstream cell if the third argument of `whenNext` is `None` or `w` if the third argument of `whenNext` is `Some(w)`. A `WhenNextComplete` result means that the dependent cell should receive an update, and this update should be final. Final completions are discussed in detail below. A `FalsePred` result means that the dependent cell should *not* receive an update.

In the above example, the second and third arguments of `whenNext` express the following update logic: c_2 is only updated if c_1 is updated with a multiple of 10, and in this case c_2 is updated with the same value as c_1.

2.4 Cyclic Dependencies

Using `whenNext` it is possible to create cyclic dependencies between cells. For a real-world example, consider the static analysis of JVM bytecode, and more specifically *immutability analysis* (this case study is discussed in more detail in Section 4). Roughly, immutability analysis attempts to answer the question whether a given class declaration defines a class whose instances can never be mutated after construction. The analysis can be implemented using Reactive Async by creating a cell for each analyzed class. The cell holds values taken from an "immutability lattice" with elements $\bot$ (the initial value of the cell), `ConditionallyImmutable`, `Mutable`, and `Immutable`. The value `ConditionallyImmutable` expresses the fact that the corresponding class C depends on potentially mutable types (*e.g.,* generic class `List[T]`) while C itself does not introduce mutability (*e.g.,* reassignable fields). The introduction of dependencies is straightforward: the cell of a class C depends on the cell of a class D if C "uses" D in some way, such as by declaring a field of type D (the precise details of a real-world immutability analysis are irrelevant and therefore omitted; Section 4.2 discusses more details). The general approach taken by our immutability analysis is to assume that

```
1  class X {
2    private final Y y;
3    X() {
4      y = new Y(this);
5    }
6  }
7  class Y {
8    private final X x;
9    Y(X x) {
10     this.x = x;
11   }
12 }
```

Figure 5. Immutability analysis with cyclic dependencies

a class is immutable unless the analysis finds contradicting evidence.

The example in Figure 5 shows how to produce cyclic dependencies in this context. Note that the code shown in Figure 5 is essentially a Java rendition of the equivalent JVM bytecode; in particular, `scalac` rejects the naive Scala equivalent.[2] For the shown example, our analysis (a) assigns value `ConditionallyImmutable` to each cell, given that each class does not introduce mutable elements itself, and (b) registers the following two dependencies:

```
1  cellForX.whenNext(cellForY,
2    x => if (x == Mutable) => WhenNext else FalsePred,
3    Some(Mutable))
4  ...
5  cellForY.whenNext(cellForX,
6    x => if (x == Mutable) => WhenNext else FalsePred,
7    Some(Mutable))
```

Clearly, the `whenNext` invocations create a dependency cycle. As a result, the immutability analysis so far fails to determine the correct result value, namely, `Immutable`, for the cells.

Dependency Resolution Reactive Async supports the resolution of cyclic dependencies as follows.

First, the runtime system identifies points in the program execution when it can detect dependencies between cells which can never be updated again. A state in the program execution that is suitable for this, is a state where it is guaranteed that none of the cells will be updated again. Clearly, this is the case when all threads capable of performing updates are idle and there are no outstanding tasks (performing updates) to be executed by these threads. When a thread pool reaches this state, it is said to be *quiescent*. Thus, to be able to detect quiescence of all update-performing threads, it is sufficient to ensure that *all cell updates are performed by tasks executed on a thread pool*. In turn, when this thread pool becomes quiescent, it is guaranteed that none of the cells will be updated again, unless a new task is submitted to the thread pool from a non-pool thread.

[2] Patterns such as the one shown here *are* relevant in practice, given occurrences in widely-used software like JDK version 8.

```
trait Key[V] {
  def resolve[K <: Key[V]](cells: Seq[Cell[K, V]]):
        Seq[(Cell[K, V], V)]
  def fallback[K <: Key[V]](cells: Seq[Cell[K, V]]):
        Seq[(Cell[K, V], V)]
}
```

Figure 6. The Key trait for specifying resolution strategies

Second, we need to distinguish between final and non-final values of cells. In the above example, each cell initially receives value `ConditionallyImmutable`. If this value would *not* be expected to change any more, then cycle resolution would not be necessary, since `ConditionallyImmutable` would be a satisfactory final result value. However, in the case of the analysis, a value should only be considered "final" if no other value would be allowable (otherwise, the analysis could return incorrect results, as explained above). Clearly, in the above example, another value, namely `Immutable`, is allowable, so the `ConditionallyImmutable` values should not be considered final. Non-final cell values then enable Reactive Async to detect cycles: cells with non-final values need an extra resolution step when the underlying thread pool reaches quiescence, since there are no outstanding updates that would complete such cells with a final value. Thus, when the thread pool becomes quiescent, all cells with non-final values are checked to see if they form part of a *dependency cycle*. Dependency cycles, more precisely closed strongly-connected components (*c*SCC), are determined and then a custom resolution strategy applied to all cells of a cSCC.[3] Resolution strategies are specified using *keys*, another abstraction that, like lattices, form part of the "configuration" of a cell.

Keys A *key* specifies the resolution strategy that is to be used for a group of cells. Essentially, a concrete key implements methods for resolving cyclic dependencies. Figure 6 shows the `Key` trait with its two methods for specifying resolution strategies. The first method, `resolve`, takes an entire cSCC as an argument (as a Scala sequence), and returns a sequence of pairs where each pair contains a cell as well as the value with which the cell should be resolved. (The actual resolution is done by the runtime system, which schedules appropriate cell updates.)

Fallbacks Interestingly, the mechanism used by Reactive Async to detect unresolved, cyclic dependencies enables *fallbacks*, which increase the expressiveness of the programming model further. In the case where in a state of quiescence there are cells without final values and *without dependencies on other cells*, dependency resolution as explained above is not applicable. To handle also this case, keys allow specifying *fallback values* for cells affected in such a way. A key specifies fallback values by implementing the `fallback` method shown in Figure 6. The method receives all cells that have neither final values nor dependencies, and returns a sequence of pairs where each pair contains a cell and a value with which the cell should be resolved. This may subsequently lead to the completion of those cells which have outgoing dependencies, but which do not take part in cyclic dependencies.

[3] A closed SCC is an SCC where each node belonging to it is only connected with nodes in the SCC, *i.e.,* there are no outgoing dependencies.

```
sealed abstract class Try[+T] { ... }
final case class Failure[+T](exception: Throwable)
                 extends Try[T] { ... }
final case class Success[+T](value: T)
                 extends Try[T] { ... }
```

Figure 7. Type `Try[T]`

3. Implementation

The implementation of Reactive Async consists of two main components: the cell and the handler pool. The former implements the `Cell` and `CellCompleter` interface traits presented in Section 2. The latter implements (a) concurrent task execution, (b) quiescence detection, and (c) dependency resolution.

Cell Similar to a promise, a cell can be in one of two states: (a) completed with a final result, or (b) not completed.

The state of a completed cell of type `Cell[K,V]` is simply a value of type `Try[V]`, shown in Figure 7.[4] A value of type `Failure[V]` is reserved for the case where a cell is completed using an expression which throws an exception. A value of type `Success[V]` indicates a successful completion with a value of type `V`.

A cell that has not been completed maintains an *intermediate result* as well as a set of *dependencies*. For a cell of type `Cell[K,V]`, the intermediate result is an element of a lattice implemented by a type class instance of type `Lattice[V]`. There are two kinds of dependencies: *onNext* dependencies and *onComplete* dependencies.

A cell c_1 with an *onNext* dependency on a cell c_2 may be updated whenever c_2 is updated using `putNext` (*onComplete* dependencies are analogous). Recall the example from Section 2.3:[5]

```
c1.whenNext(c2,
  x => if (x % 10 == 0) WhenNext else FalsePred,
  None)
```

The above `whenNext` invocation *dynamically* introduces an *onNext* dependency of cell c_1 on cell c_2. Whenever c_2 receives a monotonic update, this dependency triggers an evaluation of the predicate provided in the `whenNext` invocation, by submitting a corresponding callback for execution on the associated handler pool. c_1's intermediate result is

[4] Type `Try[T]` is part of Scala's standard library in package `scala.util`.

[5] Note that c_1 and c_2 are swapped.

then updated according to the evaluation result (WhenNext or FalsePred). To keep the overhead of triggering such an update low, the corresponding callbacks are maintained as task objects as part of their dependencies.

It is important to delete dependencies as soon as they are no longer needed. This is particularly important for cases where cells are kept accessible for longer periods of time. For example, in our case studies (see Section 4) cells are used to represent code entities for the purpose of static analysis. Here, code entities and their cells live "forever." As a result, it is crucial to free up objects implementing dependencies as early as possible to reduce the risk of out-of-memory errors.

The key insight for effectively removing unneeded dependencies is the fact that when a cell is completed with a final result (*e.g.,* using putFinal), all its *onNext* and *onComplete* dependencies can be removed, since further updates are no longer possible. However, it is important that this removal is efficient, since each invocation of putFinal triggers a removal. This is done by having each dependency maintain references to both the source and the target cell, as well as storing callbacks in hash tables indexed by cell.

Handler Pool The second main component of the implementation of Reactive Async is the *handler pool*. A handler pool is a thread pool with extensions for *quiescence detection* and *dependency resolution*. A handler pool allows registering callbacks which are executed asynchronously whenever the handler pool reaches quiescence:

```
def onQuiescent(handler: () => Unit): Unit
```

Quiescence detection is implemented using an atomically-updated reference to an instance of the following class:

```
class PoolState(
        val handlers: List[() => Unit] = List(),
        val submittedTasks: Int = 0)
```

Note that PoolState is not a case class, since an atomic reference of type AtomicReference[PoolState] requires PoolState to implement by-reference equality.

Cyclic dependencies are resolved when the handler pool is quiescent. Dependency resolution first determines all closed strongly-connected components (cSCCs) of incomplete cells. For each cSCC, the resolve method of the corresponding Key (see Figure 6) is invoked to determine the values that the affected cells are completed with.

4. Case Studies

Our case studies aim at evaluating the following research questions:

- Is our approach sufficiently expressive to implement complex computations that require managing cyclic dependencies?
- Is the performance of our approach competitive compared to a hand written implementation that achieves the same expressivity at a lower level of abstraction?

We performed two case studies to evaluate and demonstrate the benefits of using reactive async for the implementation of static analyses. The first case study is the implementation of a basic analysis for identifying pure methods (*Purity Analysis*). The second one determines the mutability of instances of some class (*Immutability Analysis*). Both are implemented using the static analysis framework OPAL [9][6] and both have functionally equivalent implementations, which use OPAL's Fixed-Point Computations Framework (called *FPCF* in the following). Compared to Reactive Async, FPCF provides an API that offers similar features as Reactive Async, but which is build around traditional locks and monitors. Furthermore, FPCF ensures that all user-defined computations w.r.t. a specific value – captured by a cell in Reactive Async – are never executed in parallel. For example, if – in case of Reactive Async – a value of a specific cell depends on the values of multiple other cells, then the user predicate(s) which is(are) registered using whenComplete may be executed in parallel. In case of the FPCF all computations related to a value are executed sequentially.

For the evaluation we always analyzed the complete Mac OS X (10.11.6) Oracle JDK 8 update 45 on a Mac Pro with a 3 GHz, 8-core (16 hyperthreads) Intel Xeon E5 with 32 GB RAM; the JVM was given 24 GB of heap memory. The performance data is visualized using box plots where the whiskers represent the min/max values (Figure 8 and 9).

4.1 Purity Analysis

Here, we consider those methods as being pure that – during one instantiation of the program – always return the same value given the same input values. The implemented analysis rates those methods as being pure which only perform basic arithmetic operations and non-virtual method calls of pure methods. If the method performs any of the following operations then the method is considered to be impure: array related operations, virtual method calls, calls of native methods, synchronization or instance-based field accesses. Hence, this analysis is not a full-blown purity analysis (*e.g.,* as presented in [10, 24]), but it captures the essential part: a method's purity depends on the purity of other methods; even if the methods are in a cyclic calling dependency.

The first step of the analysis is to associate each method with one cell (completer) to store the purity information. After that, the analysis runs in parallel for all methods and tries to determine for each method whether it is pure or impure by analyzing the method's body. For example, if the analysis only sees arithmetic and control-flow instructions – such as in Math.abs(Int,Int) – it immediately completes the method's cell using putFinal(Pure). If the analysis on the other hand sees, *e.g.,* an array access, it instead completes the method's cell using putFinal(Impure). In case the analysis encounters a non-virtual method call the analysis may still

[6] See http://www.opal-project.de.

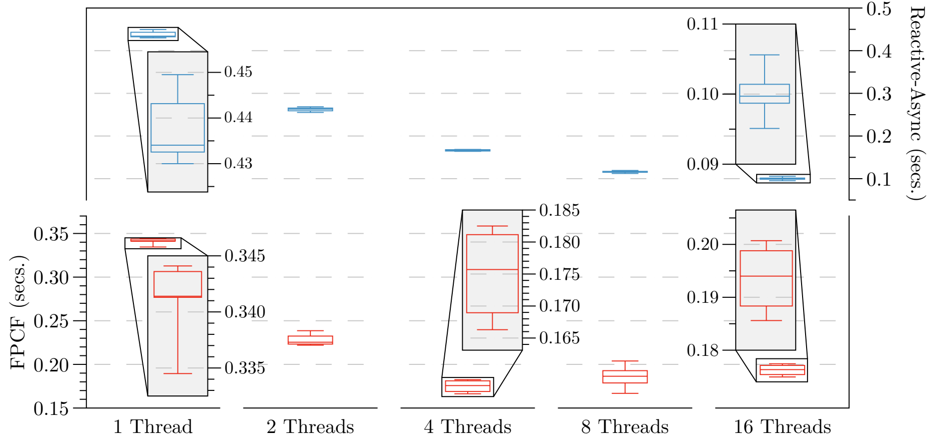

Figure 8. Runtime of the purity analysis for the entire JDK 8 given a fixed number of threads.

be able to subsequently determine that the method is impure, but it can no longer determine if it is pure without taking the purity of the called method into consideration. To do that it immediately calls the method `whenComplete` on the target method's cell completer to register a handler:

```
callerCell.whenComplete(targetCell, _ == Impure,
        Some(Impure))// complete this cell using Impure
```

This handler will then complete the cell of the calling method using `Impure` if the called method's cell is completed correspondingly; otherwise it does nothing. Hence, for methods which call other methods the analysis will never immediately complete the calling method's cell using `Pure`. Instead, it will create a dependency on the target method's cell. This design, which immediately propagates `Impure`, ensures that all impure methods are guaranteed to be identified before we reach quiescence. Later, when all computations have finished and the state of quiescence is reached, the framework will then use the fallback value (`Pure`) to complete the empty cells which are in a cyclic dependency or which have no more dependencies.

Overall, 9 out of 113 lines of code ($\approx$ 8%) are related to querying and completing (other) cells. In case of FPCF 48 out of 166 lines of code ($\approx$ 29%) are directly related to the interaction with the framework. The performance data is seen in Figure 8 and shows the time to analyze all 270 453 methods of the JDK using different numbers of threads. As shown, the FPCF framework performs better than Reactive Async when we just use one or two threads, but performs worse after that. Additionally, it can be seen that in case of Reactive Async using more threads always leads to a performance improvement while using more threads in case of FPCF only initially improves the performance. In both cases using more than 16 threads leads to a performance decrease (not shown). Overall, when we compare the two best settings – 4 threads in case of FPCF and 16 threads in case of Reactive Async – the latter is $\approx$ 1.75 times faster.

4.2 Immutability Analysis

This analysis determines the immutability of instances of a class by analyzing the immutability of its instance fields. The analysis distinguishes the three cases explained next: *Immutable* means that the instances of the class as well as all directly and indirectly referenced objects are immutable. *Conditionally Immutable* is used for those objects whose fields are final, but where the transitive closure might reference objects where the fields can be updated. Typically, immutable collection classes are only conditionally immutable. All classes that are neither immutable nor conditionally immutable are *Mutable*.

As in the previous case, this analysis is also counter-example driven. If the class defines non-final fields, then the class is considered to be *mutable* and the analysis of the class finishes. If all fields are final then the declared field types are further analyzed. In case of a primitive type, the field is treated as immutable. For referenced typed fields the immutability of the respective type is used as the field's immutability. In case that any field is *mutable*, the declaring class is considered as being *conditionally immutable*. Similarly, if a field is *conditionally immutable* then the class is also only conditionally immutable. Naturally, the immutability of a type – as required when analyzing the immutability of fields – is the join of the immutability ratings of all subtypes. Additionally, the immutability of objects of a class is always constrained by the immutability of instances of the superclass. E.g., if instances of a superclass are mutable, so are all instances of its subclasses.

The implementation of the analysis creates two cells (completers) for each class. One to store the information about the immutability of the objects of the class (`OI`) and one to store the defined type's immutability (`TI`). The later abstracts over all subtypes. After the creation of the cells, the analysis iterates over all class files in parallel and tries to determine the immutability of its instances by analyzing the

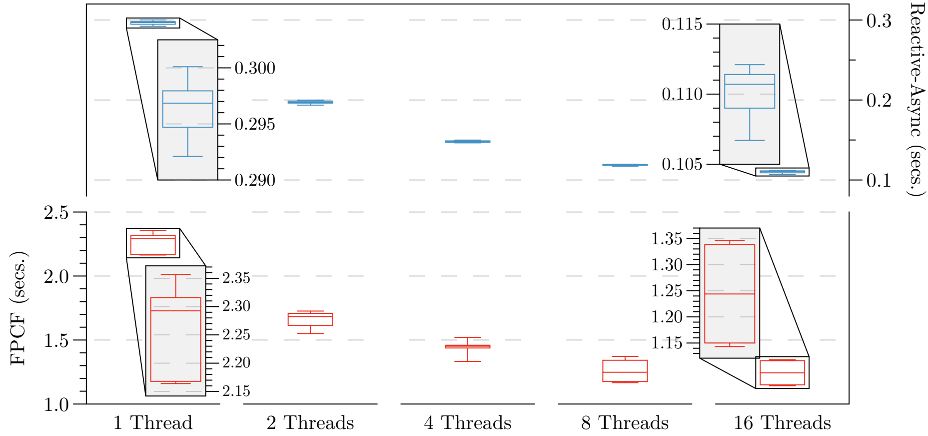

Figure 9. Runtime of the immutability analysis for the entire JDK 8 given a fixed number of threads.

fields as described. If a non-final instance field is found the `OI` cell is immediately completed using *mutable*; in case of a reference-typed field a callback is registered with the corresponding `TI` cell which sets the `OI` cell to mutable or conditionally immutable when the `TI` cell is completed accordingly. To complete a `TI` cell a callback is registered with all `TI` cells of all subtypes unless the type is a leaf type. In that case a callback is registered with the corresponding `OI` cell. Hence, as in case of the previous example, the analysis is very simple, but the complete infrastructure is implemented. Compared to the previous case the number of dependencies is very high.

The Reactive Async based implementation is basically one analysis which is 294 lines long; the FPCF-based implementation is split into two analyses and is $253 + 171 = 424$ lines long. In both cases basically all code directly interacts with the respective framework, since the analyses are just simple loops over all fields. As can be seen in Figure 9, Reactive Async is $\approx 10x$ faster than FPCF, which is due to the high amount of locking that needs to be done by FPCF.

4.3 Conclusion

As both studies have shown, using Reactive Async makes it possible to – at least – express core aspects of classical static analyses in a much more concise manner when compared to an API which uses traditional locks and a corresponding programming model. It has been shown that explicit support for resolving cyclic computations significantly benefits corresponding computations; in both cases, no code was written to handle cycles. Using other state of the art deterministic concurrent programming models would have required the developer to explicitly handle the situation. W.r.t. the performance the case studies have shown that Reactive Async scales very well and enables efficient use of multi-core CPUs.

5. Benchmarks

We also evaluate the performance and scalability of Reactive Async using two benchmarks, *Parallel Sum* and *Monte Carlo Net Present Value*, which we describe below. For each benchmark we compare three different implementations: a sequential implementation, a concurrent implementation based on Scala's futures and promises, and a concurrent implementation based on Reactive Async. All performance measurements are based on the same hardware and software setup as the case studies (see Section 4).

Parallel Sum This benchmark computes the sum of a large collection of random integers in parallel. To expose sufficient parallelism, we use a collection of size $2^{24} = 16777216$. Furthermore, to enable efficient splitting of the collection we use Scala's `Vector[Int]`, which is backed by a bit-mapped vector trie with a branching factor of 32.

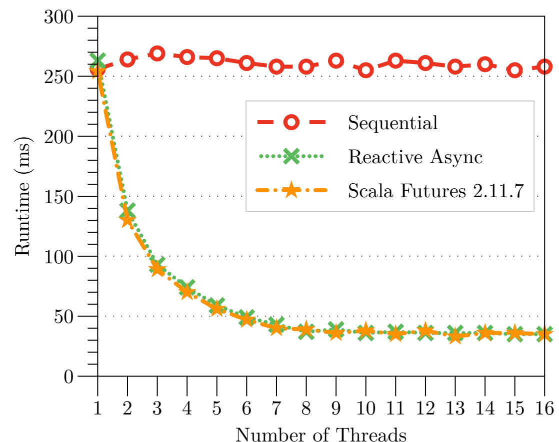

Figure 10. Runtime of the *Parallel Sum* micro benchmark in relation to the number of used threads

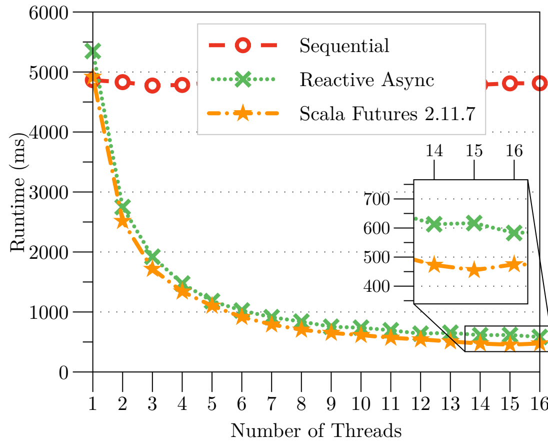

Figure 11. Runtime of the *Monte Carlo Net Present Value* benchmark in relation to the number of used threads

Figure 10 shows the runtime of our implementations in relation to the number of used threads. The performance of Reactive Async is virtually identical to Scala's futures in this benchmark. The speed-up compared to the sequential implementation is about 1.91x for 2 threads (futures: 2.03x), about 6.97x for 8 threads (futures: 6.62x), and about 7.37x for 16 threads (futures: 7.37x).

Monte Carlo Net Present Value This benchmark computes the Net Present Value[7] while allowing for uncertainty in the discount rate and the cash flows by using a Monte Carlo simulation. Our implementation is a Scala port of an open-source Java implementation by Alan Hohn,[8] which is based on Doug Lea's fork/join framework [16].

Figure 11 shows the runtime of our implementations in relation to the number of used threads. Overall, the observed performance trends are similar to the *Parallel Sum* benchmark. However, we can see that Scala's futures are slightly more efficient. For 14–16 threads, futures are between 23% and 36% faster than Reactive Async. Two main reasons are likely: first, the *Monte Carlo Net Present Value* benchmark requires more synchronization than the *Parallel Sum* benchmark; as a result, the more complex state updates within cells become observable. Second, Scala's futures have undergone performance tuning by industry experts, whereas the same level of optimization effort has not been applied to the current implementation of Reactive Async.

Summary The presented micro benchmarks show that Reactive Async exhibits a scalability competitive with Scala's futures, a finely-tuned, state-of-the-art implementation of futures on the JVM. In terms of runtime, Scala's futures are up to 36% faster than Reactive Async in our micro benchmarks. We expect this percentage to decrease with an effort to carefully optimize the implementation of Reactive Async.

[7] See `https://en.wikipedia.org/wiki/Net_present_value`.

[8] See `https://github.com/AlanHohn/monte-carlo-npv`.

6. Related work

Programming Models for Concurrency Control Over the years researchers have proposed a number of advanced concurrency control mechanisms. Software Transactional Memories (STM) [12] handle the problem of concurrent access to shared data introducing abstractions that define the boundaries of a transaction. Transactions abort in case of a conflict (optimistic control) or check not to introduce conflicts in advance (pessimistic control). Actors [13] encapsulate state and control, reacting to asynchronous messages. Since actors do not share memory, race conditions are not possible by construction. Imperative languages pose the additional challenge that races can arise on messages. The Async/Finish model [7] provides high-level, lexically-scoped constructs for creating parallel tasks and waiting for their termination. This approach simplifies static analysis and enables compiler optimizations. Finally, additional models have been derived via the combination of other models. For example, Habanero-Scala (HS) [14] is a Scala library that supports a hybrid programming model, combining the Async/Finish model and the Actor model. In contrast to our approach, none of these solutions guarantee determinism.

Deterministic Concurrent Models Isolation types [6] are a mechanism for parallel execution of application tasks. Isolation types are used for data shared among tasks. Tasks fork and join isolated revisions: tasks read and modify their own private copy of the shared data. Copies are created and merged automatically, conflicts are solved deterministically, in the way declared by each isolation type.

Programming models for deterministic concurrency guarantee the equivalence to a sequential execution [3, 5, 21, 25]. They restrict the tasks to avoid conflicts that cannot be automatically resolved via the type system, or at runtime via blocking (optimistic) or aborts and retry (optimistic).

FlowPools [23] provide a deterministic concurrent dataflow abstraction for multisets, with an efficient lock-free implementation. Reactive Async generalizes FlowPools by supporting arbitrary lattice-based data, including but not limited to multisets. Furthermore, Reactive Async supports the resolution of cyclic dataflow dependencies, whereas FlowPools are restricted to acyclic dataflow graphs. LVars [15] is a deterministic-by-construction parallel programming model based on shared state similar to Cells: they take the least upper bound of the old and new values with respect to the lattice to guarantee determinism. Similar to `putFinal`, LVars can be freezed after a write, to ensure that no further values can be added. In contrast to our approach, LVars do not support cyclic dependencies and do not address the problem of fallback computations.

In distributed systems, monotonicity has been used to guarantee eventual consistency. Bloom [1] allows programming distributed systems that are eventually consistent based on monotonically updated data collections.

DPJ [4] introduces an effect system for deterministic concurrency in Java. The compiler checks that memory *regions* are linear, avoiding concurrent accesses by multiple writers. In contrast to our approach and solutions like LVars, which are based in monotonic writes, DPJ achieves determinism by restricting *when* reads and writes can happen.

Reactive Programming Reactive programming [2] is a programming paradigm that aims at supporting reactive applications with dedicated language abstractions (*e.g.,* event streams, signals). Languages like ELM [8], Scala.react [18], and ReactiveX/Rx [19] support asynchronous execution of reactive computations. Similar to cells, reactive abstractions support the composition of asynchronous computations. However, while some of the existing reactive programming languages support dynamic dependencies [20], they do not provide determinism nor support cycles in the dependencies between reactive values.

7. Conclusion

Restricted expressivity has been a long-standing challenge for deterministic concurrency. We propose a concurrent programming model, *Reactive Async*, that overcomes several limitations of existing solutions. It allows developers to write composable, deterministic concurrent code supporting both cyclic and dynamic dependencies. Case studies in the domain of static analysis show that *Reactive Async* can easily tackle problems where existing solutions are not effective due to a lack of expressivity. Performance evaluations on both case studies and benchmarks show that our implementation allows writing faster applications compared to hand-written concurrent code and to retain the scalability of state-of-the-art implementations of futures.

Acknowledgments

This work is partially supported by the European Research Council, grant No. 321217.

References

[1] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, pages 249–260, 2011.

[2] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, pages 81–96, 2009.

[4] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic by Default. In *HotPar*, page 4, 2009.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, pages 97–116, 2009.

[6] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, pages 691–707, 2010.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[8] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422, 2013.

[9] M. Eichberg and B. Hermann. A software product line for static analyses: The OPAL framework. In *SOAP*, pages 1–6, 2014.

[10] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *CCS*, pages 161–174, 2008.

[11] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Futures and promises. `http://docs.scala-lang.org/overviews/core/futures.html`, 2012.

[12] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, May 2007.

[13] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[14] S. M. Imam and V. Sarkar. Integrating task parallelism with actors. In *OOPSLA*, pages 753–772, 2012.

[15] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. In *POPL*, pages 257–270, 2014.

[16] D. Lea. A Java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[17] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, pages 260–267, 1988.

[18] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, pages 707–731, 2013.

[19] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[20] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. In *OOPSLA*, pages 1–20, 2009.

[21] P. Pratikakis, J. Spacco, and M. Hicks. Transparent Proxies for Java Futures. In *OOPSLA*, pages 206–223, 2004.

[22] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *PPOPP*, pages 151–160, 2012.

[23] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173, 2012.

[24] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[25] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–453, 2005.