Pluggable Scheduling for the Reactor Programming Model

Aleksandar Prokopec
Oracle Labs, Switzerland
aleksandar.prokopec@oracle.com

Abstract

The reactor model is a foundational programming model for distributed computing, whose focus is modularizing and composing computations and message protocols. Previous work on reactors dealt mainly with the programming model and its composability properties, but did not show how to schedule computations in reactor-based programs.

In this paper, we propose a pluggable scheduling algorithm for the reactor model. The algorithm is customizable with user-defined scheduling policies. We define and prove safety and progress properties. We compare our implementation against the Akka actor framework, and show up to $3\times$ performance improvements on standard actor benchmarks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.4.1 [*Process Management*]: Scheduling

General Terms Algorithms

Keywords reactor model; reactors; scheduling; composable distributed computing; event streams; channels; actors

1. Introduction

The recently proposed reactor model (Prokopec and Odersky 2015) (Prokopec 2016) uncovered a new route to composable distributed computing. Instead of composing message protocols across multiple actors, the reactor model advocates protocol composition within a single unit of concurrency called a reactor. This composition is achieved by exposing multiple typed first-class event streams instead of a static `receive` statement.

The original reactor model proposal (Prokopec and Odersky 2015) dealt with the programming model, but did not discuss the underlying implementation. Existence of multiple event streams within each reactor poses a scheduling problem that differs from that of the standard actor model, in which each actor has a single mailbox. In the reactor model, the fundamental constraint is the following: events from different event streams must be scheduled fairly, but serially for any two event streams belonging to the same reactor.

The goal of this paper is twofold. First, we propose a scheduler for the reactor model, identify its properties and show correctness. Second, we make the scheduler pluggable, allowing clients to implement custom scheduling policies.

There are several reasons why a scheduler should be pluggable. First, it is expensive and time consuming to develop an optimal scheduler. A more prudent plan is to develop a system with a sub-optimal scheduler, and then (let clients) improve it incrementally when concrete requirements arise.

Second, not every scheduler is perfect for every situation. A scheduler can claim optimality across all possible workloads, i.e. be Pareto optimal, but it is likely that there is a more optimal scheduler for a particular class of workloads. For example, in the Ping-Pong benchmark (Imam and Sarkar 2014a), another message is likely to arrive soon after the actor sends a message, and it helps to keep a (re)actor activated even when there are no pending messages to handle. However, in the Thread Ring benchmark, the same heuristic wastes processor time, as it is unlikely that a message will arrive soon. In these cases, users should be able to decide which scheduling policy is appropriate for their workload.

Third, certain scheduling policies are application-specific and rely on explicit domain knowledge. For example, if a reactor needs a special system-wide resource (such as a GPU or a DSP), then the scheduler needs to negotiate the availability of the resource with the OS. A generic scheduler does not have OS-specific knowledge, and this warrants a user-defined scheduling policy.

This paper brings forth the following contributions:

- Detailed description and implementation of a scheduling algorithm for the reactor programming model.
- A pluggable mechanism for user-defined scheduling policies, which can embed application-specific knowledge.
- A list of safety and progress properties for a reactor scheduler to satisfy. We analyze the proposed algorithm, and show that it satisfies these properties under specific assumptions on the user-defined scheduling policy.
- A comparison on the Savina benchmark suite (Imam and Sarkar 2014a) with the Akka actor framework. We show that our reactor implementation outperforms Akka on 6

AGERE'16, October 30, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4639-9/16/10...$15.00
http://dx.doi.org/10.1145/3001886.3001891

out of 8 benchmarks by a factor of $1.1 - 3.3\times$, and otherwise has comparable performance.

This work focuses on scheduling reactors in a single reactor system, on a single shared-memory machine. Scheduling reactor execution in a fault-tolerant distributed setting is not the goal of this work. That problem is based on an entirely different set of assumptions (such as faults, preemptions, network delay, lack of shared memory), and consequently results in different abstractions. In practice, this is the task of the cluster manager, and not the reactor scheduler. However, effective single-machine scheduling is a performance prerequisite for efficient distributed computations.

Code examples are written in Scala (Odersky and al. 2004), a statically compiled language, primarily targetting the JVM. Syntax is similar to Java, but more concise. Variables and final variables are defined with keywords `var` and `val`, respectively, and methods with the `def` keyword, as in Python. Type annotations come after a `:` following an identifier, similar to Pascal. Function objects are declared with a list of parameters, followed by `=>` and a body. Partial functions are declared as a list of `case` statements, and are defined for the values matched by at least one of the cases. Traits and the `with` keyword are equivalents of Java interfaces and `implements`. Type parameters are enclosed in square brackets, `[]`. Operators, such as `!`, are normal methods with symbolic names. Critical sections are delimited with a `synchronized` block.

The rest of the paper is structured as follows. In Section 2, we describe the reactor model. We then describe the relevant internals of our reactor framework and the pluggable scheduling algorithm in Section 3. In Section 4, we show how our scheduling algorithm can be customized with user-defined components. Section 5 evaluates the scheduling implementation with standard workloads, Section 6 shows related work, and Section 7 concludes.

2. Reactor Model

The reactor programming model (Prokopec and Odersky 2015) is a generalization of the standard actor model (Erlang 2015) (Agha 1986). There are three major differences between these two models. First, the reactor model exposes multiple first-class event streams instead of a static `receive` statement. Second, in the reactor model, a computation can wait for events from multiple event streams simultaneously, whereas an actor can be suspended on a single `receive` statement at a time. Third, targets of message sends are typed channels instead of untyped actor references[1]. As argued before (Prokopec and Odersky 2015), these three fundamental differences allow modularity and composition of message protocols within a single reactor, a feature that was previously not possible with actors alone. For example, the reactor model allows defining best-effort and reliable broadcasts, failure detectors (Guerraoui and Rodrigues) and CRDTs (Shapiro et al. 2011), and exposing them as reusable components, which can be either embedded into a reactor, or further composed into more complex protocols.

In the reactor model, the principal unit of concurrency is called a *reactor*. Analogous to how an actor can process at most a single message at once, a reactor can process at most a single event at any point in time. This *serializability* property is one of the major strengths of (re)actors, as it allows users to access local state without the use of synchronization.

Consider a reactor that counts how many events it received. The following code snippet declares a reactor template `AnalysisReactor` that tracks how many string events it received. Field `numEvents` is part of the reactor's state:

```
class AnalysisReactor extends Reactor[String] {
  var numEvents = 0 }
```

Defining a reactor template does not yet start a reactor instance. Before we see how to do that, we need to define how the reactor receives events. Entities that allow handling incoming events are called *event streams*. Every reactor gets a default event stream called `main.events` when it is created. To receive an event, users need to pass an event handler to the stream's `onEvent` method. We extend the body of the previous reactor template with a call to `onEvent`:

```
main.events.onEvent { x => numEvents += 1 }
```

Generally, an event stream has the type `Events[T]`. In our case, `main.events` has the type `Events[String]`, because we declared a reactor of type `Reactor[String]`.

A reactor is not limited to receiving events on a single event stream. During its lifetime, a reactor can receive from any number of event streams. Apart from the main event stream, every reactor has a system event stream called `sysEvents` that delivers lifecycle events – for example, when the scheduler assigns execution time to the current reactor, or the reactor terminates. We can react to a subset of system events by passing a partial function to `onMatch` method of the event stream. In the following, we expand the earlier reactor template with a variable `numSch`, and count the number of times the reactor was assigned execution time. We expect that each time the reactor is scheduled, it handles several events. When the reactor terminates, we print the average number of events handled each time it got scheduled:

```
var numSch = 0
sysEvents.onMatch {
  case Scheduled => numSch += 1
  case Terminated => print(numEvents / numSch)
}
```

Every event stream has a corresponding *channel*. A channel is the writing end of the event stream. It has the type `Channel[T]`, where `T` corresponds to the event stream type. Whereas an event stream can be used only by the reactor that owns it, a channel can be shared with any other reactor. The basic operation on a channel is an event send `!`. In the following, we extend the reactor template to send a message to the main channel when the reactor starts:

```
sysEvents.onMatch {
  case Started => main.channel ! "started" }
```

[1] In Erlang, actor references are called process IDs.

To create additional channels and event streams, a reactor can use the `open` statement. Given the type of events, say `Int` for integers, the `open` statement returns a fresh pair of a channel and the corresponding event stream:

```
val (numberEvents, numberChannel) = open[Int]
```

To create a running reactor instance from a reactor template, we need to call the `spawn` method of the reactor system. Spawning a reactor returns its main channel:

```
val ch: Channel[String] =
  system.spawn(Proto[AnalysisReactor])
```

`Proto` is a wrapper around the specific reactor class, used to set properties such as the textual name of the reactor.

We stated that reactors generalize actors. To support this, we encode an Akka-style (Akka 2015) actor with a reactor from the Reactors.IO framework (Prokopec 2016). The reactor receives events of type `Any`, which is the top type in Scala. Any event `x` from the main event stream is forwarded to the partial function `receive`. Otherwise, the event is discarded.

```
abstract class AkkaActor extends Reactor[Any] {
  def receive: PartialFunction[Any, Unit]
  main.events.onEvent { x =>
    if (receive.isDefinedAt(x)) receive(c)
  } }
```

Exact formal semantics of the reactor model can be found in related work (Prokopec and Odersky 2015). In a nutshell, the reactor model has the following components:

- **Starting computations:** reactor templates that define reactors, and the `spawn` method used to start them.
- **Receiving events:** event streams and the `onEvent` method, used to subscribe to incoming events.
- **Sending events:** channels and the `!` operator, used to send events to other reactors.
- **Modularising protocols:** the `open` method, used to create supplementary channels in the current reactor.

The main difference with the actors is that there are multiple event streams in each reactor, and events can be delivered on any of them. Before examining the proposed scheduling algorithm, we examine its desirable properties.

2.1 Properties of a Reactor Scheduler

We now explore some desirable properties that a reactor scheduler should satisfy. In what follows, we say that an event is *delivered* if it is enqueued on an event queue. We say that a reactor is *activated* when it becomes scheduled to process some of the delivered events. We say that an event is *handled* when the event handlers from corresponding event streams get invoked for that event.

Serializability states that a reactor at any point in time runs at most one of its event handlers. Processing events serially, in sequence, prevents race conditions that would otherwise result from simultaneously manipulating reactor state. Importantly, serializability applies to events received on all event streams of the same reactor – at most one handler across all event streams may be active at any time.

Liveness states that if an event is delivered to the reactor on some event stream, then the corresponding event handler is eventually invoked. Liveness prevents starvation – a scenario where specific subsets of events are never processed, preventing normal program progress.

Although liveness ensures that all delivered events are eventually handled, it is in practice useful that a scheduler provides a stronger guarantee. A scheduler should avoid a scenario in which a set of events delivered to one event stream grows indefinitely. This can, for example, occur in a multiple producer, single consumer setting. Liveness only ensures that the single consumer is eventually scheduled, but does not prevent its event queue from growing indefinitely. To be fair, a scheduler must ensure that some event streams get processed more often than others. We formulate *fairness* as follows – for any two events x and y, such that x is the d_x-th event delivered globally and y is d_y-th, and x is the h_x-th event handled globally and y is h_y-th, difference $h_x - h_y$ must be bound by $d_x - d_y + C$, where C is a constant. This is essentially a global relaxed FIFO condition.

Aside from being fairly executed, reactors must be able to exploit parallelism. A reactor scheduling system is *scalable* if it meets the following. First, event handling must retain the same performance in the presence of concurrent event delivery. Second, event delivery time must be $O(1)$ when there are P events delivered concurrently on any subset of event streams. Third, event delivery time must be $O(1)$ irrespective of the number of event streams E in the system.

The scheduling sytem must be *efficient* – the time spent scheduling must be amortized by the execution of user code. Scheduler overhead should not be noticeable. This property is validated through an empirical evaluation.

The last important concern is *pluggability*. Clients that possess domain knowledge must be able to apply this knowledge to their scheduler to make the system more efficient. Pluggability allows controlling when a specific reactor is executed, and how much execution time it receives.

Some of these properties, such as serializability, ensure that a program never violates semantics of the reactor model. We refer to them as *safety* properties, as they guarantee that nothing bad happens. Other properties, such as liveness, fairness and scalability, improve *progress* of a reactor-based program. Their absence can in worst case prevent the program from completing, but does not violate semantics or cause incorrect behavior. As we will see in Section 3, the proposed pluggable scheduling system enforces safety properties. Progress properties are good-to-have, but not essential for all programs. For such properties, the scheduling system establishes a well-defined foundation, and delegates the decision of fulfilling them to other components.

3. Scheduling System

In this section, we describe the proposed pluggable scheduling algorithm. We start with the internals of our reactor system implementation, and then show the algorithm itself. Finally, we prove that the algorithm satisfies serializability and liveness, and is fair under specific assumptions.

3.1 Reactor System Internals

An *event queue* contains a set of delivered, but not yet processed events for a particular event stream. Since events must be handled serially within a reactor, an event queue serves as a buffer between the reactor and the senders. An event queue is an equivalent of an actor mailbox.

In the following, we show the `EventQueue` trait. Method `enqueue` atomically enqueues an event to the event queue and returns queue size. It can be called concurrently. The `dequeue` method atomically removes an event, emits it on an event stream `events`, and returns the number of remaining elements at the point when the event was removed. Method `dequeue` is quiescently consistent – it can be called by at most a single thread at a time. When `dequeue` emits the event on the associated event stream, control goes from the scheduler to handlers installed by the user code.

```
trait EventQueue[T] {
  def enqueue(x: T): Int
  def dequeue(): Int
  def events: Events[T] }
```

A connector of type `Connector[T]` is a wrapper that binds an event stream, a channel and an event queue together. Calling `open` creates a new connector.

Different reactors have different textual names, used to retrieve their channels. The set of all possible names comprises the *namespace* of the reactor system. At any point in time, at most a single reactor can have any single name.

When created, every reactor is assigned a unique numeric ID. The set of all possible UIDs forms the *UID space*. During the entire lifetime of the system, every UID can be assigned to at most one reactor, and cannot be reused.

A *reactor system* is an entity that contains a set of reactors, the scheduling system, and a single namespace and UID space. Usually, there is a single reactor system per process, but users can create additional reactor systems if necessary. Configuration properties such as pickling and network resources are set when creating the reactor system.

Prototype, represented with the `Proto[T]` type, is a configurable wrapper around the reactor template. It allows configuring the textual name and the scheduling policies of the reactor instance, and is passed as an argument to `spawn`.

Immediately before the reactor instance starts, the reactor system creates a *frame* object of type `Frame`, used to hold internal reactor state – reactor name, UID, scheduling policy, connectors, lifecycle state and information on whether the respective reactor is currently executing.

A reactor's scheduling policy is captured in a `Scheduler` object. The `initSchedule` method is invoked once when the reactor is created, and `schedule` is invoked every time a reactor is activated. Method `newPendingQueue` creates a queue with a list of active connectors, and allows the scheduler to implement a queuing policy.

```
trait Scheduler {
  def initSchedule(f: Frame): Unit
  def schedule(f: Frame): Unit
  def newPendingQueue(): Queue[Connector[_]] }
```

```
1  def spawn[T](
2    system: ReactorSystem, proto: Proto[T]
3  ): Channel[T] = {
4    val uid = system.reserveId()
5    val uname = system.acquire(proto.name)
6    val f = new Frame(uid, uname, proto, system)
7    try {
8      f.active = false
9      f.scheduler = proto.scheduler
10     f.lifecycleState = Fresh
11     f.connectors = new Map[String, Connector[_]]
12     f.pending = f.scheduler.newPendingQueue()
13     f.scheduler.initSchedule(f)
14     f.main = open(f, "main", f.queueFactory)
15     activate(f)
16   } catch { case t: Throwable =>
17     system.release(uname)
18     throw t
19   }
20   f.main.channel }
21 def activate(f: Frame) {
22   val run = f.monitor.synchronized {
23     if (!f.active) {
24       f.active = true
25       true
26     } else false }
27   if (run) f.scheduler.schedule(f) }
```

Figure 1. Reactor creation

`Queue` exposes standard operations `enqueue` and `dequeue`. Note that its implementation is different than that of an event queue. A *pending queue* stores *event queues*, while an *event queue* stores events. The two are **separate entities**.

As we will see in the next section, user-defined `Scheduler` objects allow fine-tuning how the scheduling system works.

3.2 Scheduling Algorithm

From a high-level standpoint, the algorithm works as follows. When a reactor needs to execute, the `active` field in its frame is set to `true`, and the scheduler is notified. The reactor then gets execution time. It repetitively removes an event queue from the pending queue, and calls `dequeue` on the event queue until either the scheduler tells it to stop, in which case a non-empty event queue goes back to the pending queue, or the event queue becomes empty, in which case the pending queue is polled for the next event queue.

There are two ways that a reactor can get execution time. First is when a reactor instance is created with `spawn`, and the second is when an event is delivered to a reactor. In both cases, the reactor is activated and sent for execution.

We first consider the `spawn` operation, shown in Figure 1. The method starts by reserving a UID in line 4, and the reactor name in line 5. It then creates a `Frame` object in line 6. In lines 8 through 11, frame is marked as not activated, reference to the scheduler specified in the prototype is copied, the lifecycle state is set to `Fresh`, and a connector table is created. In line 12, the scheduler's `newPendingQueue` method returns the queue data structure that will hold non-empty event queues. The scheduler is asked to optionally set a custom state object in the frame's `schedulerState` field. This

```
 1 def execute(f: Frame) = {
 2   assert(f.active)
 3   assert(f.isolationCount.compareAndSet(0, 1))
 4   try lifecycleAndProcessBatch(f)
 5   finally {
 6     var repeat = false
 7     f.monitor.synchronized {
 8       if (!f.pending.isEmpty &&
 9           f.lifecycleState != Terminated)
10         repeat = true
11       else f.active = false
12     }
13     if (repeat) f.scheduler.schedule(this)
14     f.isolationCount.set(0)
15   } }
16 def checkFresh(f: Frame) {
17   val fresh = f.monitor.synchronized {
18     if (f.lifecycleState == Fresh) {
19       f.lifecycleState = Running
20       true
21     } else false }
22   if (fresh) f.reactor = proto.create() }
23 def checkTerminated(f: Frame, forced: Boolean) {
24   val term = f.monitor.synchronized {
25     val isRunning = f.lifecycleState == Running
26     val mustTerm = f.pending.isEmpty &&
27       f.connectors.length == 0
28     if (isRunning && (forced || mustTerm)) {
29       f.lifecycleState = Terminated
30       true
31     } else false
32   }
33   if (term) f.system.release(name) }
```

```
38 def lifecycleAndProcessBatch(f: Frame) {
39   try {
40     checkFresh(f)
41     processEvents(f)
42   } catch { case t: Throwable =>
43     checkTerminated(f, true)
44   } finally checkTerminated(f, false)
45 }
46 def processEvents(f: Frame) {
47   f.schedulerState.onBatchStart(this)
48   val c = popNextPending(f)
49   if (c != null) drain(c)
50 }
51 def popNextPending(f: Frame): Connector[_] = {
52   f.monitor.synchronized {
53     if (f.pending.nonEmpty)
54       f.pending.dequeue()
55     else null
56   }
57 }
58 @tailrec def drain(c: Connector[_]) {
59   val remaining = c.queue.dequeue()
60   if (f.schedulerState.onBatchEvent(c)) {
61     if (remaining > 0 && !c.isSealed) {
62       drain(c)
63     } else {
64       val nc = popNextPending(f)
65       if (nc != null) drain(nc)
66     }
67   } else if (remaining > 0 && !c.isSealed)
68     f.monitor.synchronized {
69       f.pending.enqueue(c)
70     } }
```

Figure 2. Reactor loop

is done with the call to `initSchedule` in line 13, and the default connector is allocated in line 14.

At this point, the frame is initialized and may run. A call to `activate` in line 15 activates the frame. This method acquires the frame's lock in line 22, and checks if the frame is already active in line 23. If not, the `active` field is set to `true` in line 24. If `active` was set, the `schedule` method is called in line 27. This indicates that there is a newly activated frame that should run. The scheduler must give the reactor execution time at the earliest opportunity.

When the scheduler assigns execution time on some thread, that thread must call the `execute` method shown in Figure 2. This method starts the reactor's event loop, and has several stages. First, it prepares the reactor context – it asserts that the frame is active in line 2, and optionally sets thread-local state (not shown in the code). Then, it calls `lifecycleAndProcessBatch` to continue executing the reactor's lifecycle. After the lifecycle method completes, either exceptionally or normally, `execute` checks if the reactor should continue executing or not. Line 8 tests if there are any pending event queues with unprocessed events and the reactor did not terminate. If so, the reactor is rescheduled, and otherwise its `active` field is set to `false`.

The `lifecycleAndProcessBatch` method uses two auxiliary methods `checkFresh` and `checkTerminated`. Method `checkFresh` is called before event processing starts, and it atomically changes the state from `Fresh` to `Running`. If the state changes, it means that reactor was just started, so the `checkFresh` method needs to run the reactor constructor. Constructor must run asynchronously, and not in the `spawn` method, to ensure non-blocking semantics. The constructor is run in line 22 with a call to the prototype's `create` method.

The `checkTerminated` method similarly checks for termination, and is called after processing events. A reactor must terminate if it is in the `Running` state, its pending queue is empty, and there are no more live connectors. If the argument `forced` is set to `true`, it means that user code threw an exception, and the reactor must be terminated regardless of its execution state. When the state is atomically changed to `Terminated`, the reactor name is released in line 33. In practice, all these methods emit lifecycle events on the system event stream, but we omit them from Figure 2 for brevity.

At this point, the reactor can start handling events. The method `lifecycleAndProcessBatch` calls `processEvents`, which in line 47 notifies the scheduler that a batch of events is about to be handled. The `processEvents` method then calls `popNextPending` to dequeue a non-empty connector. If a reactor just started, it is likely that no events were yet delivered, and `popNextPending` returns `null`. In this case,

```
 1 def send[T](c: Connector[T], x: T) {
 2   val f = c.frame
 3   val size = c.queue.enqueue(x)
 4   var run = false
 5   if (size == 1) f.monitor.synchronized {
 6     f.pending.enqueue(c)
 7     if (!f.active) {
 8       f.active = true
 9       run = true } }
10   if (run) scheduler.schedule(this) }
```

Figure 3. Event send

`processEvents` simply returns. If there is a non-empty connector, `processEvents` calls the `drain`.

The `drain` method calls `dequeue` on the event queue in line 59. This releases an event on the corresponding event stream, and enters user code. After event handlers process the event, `dequeue` returns the number of remaining events at the point in time when the event was removed. It then asks the scheduler if it should continue executing events in line 60. If the scheduler decides that additional events should be batched, `drain` checks if the event queue is non-empty and calls itself tail-recursively in line 62 with the same connector. If the current event queue is empty, method `drain` pops the next non-empty connector if there is one, and calls itself recursively in line 65. If the scheduler denies processing additional events, `drain` enqueues the non-empty event queue back to the pending queue in line 69.

Using `onBatchEvent`, the scheduler can decide how many events to handle. Usually, a scheduler will handle a batch of events, to amortize the cost of setting up the reactor context.

A reactor is also activated when an event is delivered on one of its event streams. This is done by the `send` method in Figure 3, which first enqueues the event on the respective event queue in line 3. If the event queue size after calling `enqueue` is exactly 1, it means that the corresponding event stream was previosly dormant, and it became active when the event was enqueued. In this case, the reactor's lock is acquired in line 5, and the event queue is placed on the pending queue in line 6. If the reactor was not previously active, its `active` field is set to `true`, and the reactor is scheduled for execution in line 10. The `execute` method from Figure 2 is eventually invoked on some thread.

3.3 Analysis of the Scheduling Algorithm

We now state several claims about its properties. We prove that the algorithm is safe with respect to serializability. For space reasons, we skip safety properties such as exactly-once delivery. We then prove liveness and fairness, with specific assumptions about the `Scheduler`.

Theorem 1 (Safety)**.** *Assume that `schedule` executes the reactor exactly once. For a specific reactor, there is at most a single event handler executing at any point in time.*

Proof. No thread is initially running `execute`. The first call to `schedule` occurs in the `activate` method in Figure 1, and the second `schedule` occurs in the `send` method in line 10 in Figure 3. If either `activate` or `send` calls `schedule`, then the `active` field was previously `false` and was atomically set to `true` by the same thread. No other thread can call `schedule` until `execute` reaches line 13 in Figure 2.

The `execute` method calls `schedule` in line 13 only if `active` was not set from `true` to `false`. It follows that, for a specific reactor, there is always at most one thread that previously left the `active` field in the `true` state, and that thread calls `schedule`. By assumption, `execute` is called only once for every `schedule` call, and `execute` calls `dequeue` for every event only once, so it follows that there is at most a single event handler executing at any time[2]. □

Lemma 1 (Deactivation)**.** *An empty event queue is never on the reactor's `pending` queue.*

Proof. We show this inductively – claim is initially true, and no operation violates it. The pending queue is initially empty. The `send` method puts only non-empty event queues to the pending queue. Events are only dequeued by `drain` in Figure 2, and this method never puts an empty event queue back to the `pending` list. By Theorem 1, no other thread can interfere by concurrently executing `drain`. □

Lemma 2 (Activation)**.** *A non-empty event queue is either on the reactors's `pending` queue, or is put on the `pending` queue after a finite number of steps, exactly once.*

Proof. An event is delivered to the event stream in line 3 of the `send` method shown in Figure 3. If `enqueue` in line 3 returns size greater than 1, then there must exist another thread for which `enqueue` previously returned 1. If this other thread did not yet put the event queue on the pending list, then no other thread can drain that event queue (since the point in time when `enqueue` returned 1), by Lemma 1.

Consider the thread for which `enqueue` returns size 1 in line 3 of Figure 3. That thread puts the event queue to the `pending` list after a finite number of steps. Next, consider the thread that calls `popNextPending`. If the event queue is non-empty when that thread subsequently calls `dequeue` in line 59, the event queue is put back to the `pending` queue by the same thread after a finite number of steps. By Lemma 1, the queue cannot become empty before this happens. □

Theorem 2 (Liveness)**.** *Assume that `schedule` eventually executes the specified reactor, and that every event queue added to the pending queue is dequeued after calling `dequeue` sufficiently many times. Then, if an event gets delivered to an event stream belonging to some reactor, that event is eventually handled by an event handler.*

Proof. By Lemma 2, a non-empty event queue is already on the `pending` queue, or will be after a finite number of steps. By assumption, every reactor is eventually executed, and every event queue in `pending` is eventually dequeued. For each such event queue, at least one event is handled. Consequently, every event is eventually handled. □

[2] In fact, check in line 3 of Figure 2 ensures this even if `schedule` calls `execute` from multiple threads.

Fairness is achieved by the scheduling policy, so the fairness proof makes heavy assumptions on its implementation.

Theorem 3 (Fairness)**.** *Let $\mathbb{S}$ be the set of reactors for which* `schedule` *was called. Assume that the scheduler always executes the reactor from $\mathbb{S}$ with the least recent event ξ, that the* `dequeue` *call on the pending queue of the respective reactor returns the event queue that contains ξ, and that* `onBatchEvent` *returns* `true` *if the argument connector contains the most recent event in the system. Then, the scheduling is fair with respect to the previous definition.*

Proof. Under the assumptions, `dequeue` call in line 59 always returns the oldest unprocessed event, so scheduling is fair with $C = 1$ for the definition in Section 2.1. □

4. Scheduling Policies

In this section, we go over implementations of the trait `Scheduler`. There are several different ways in which a `Scheduler` governs the scheduling policy. First, it decides when to execute frames submitted with the `schedule` method. Second, it decides how long a scheduled reactor should execute with `schedulerState`. Third, it decides which event stream to flush with the `newPendingQueue` method.

The `schedulerState` exposes methods `onBatchStart` and `onBatchEvent`. Most schedulers use some variant of the following `DefaultState`, which handles up to `BATCH_SIZE` events during one scheduled frame execution.

```
class DefaultState extends State {
  private var batch = 0
  def onBatchStart() { batch = BATCH_SIZE }
  def onBatchEvent(c: Connector[_]) = {
    batch -= 1
    batch > 0 } }
```

The `newPendingQueue` method decides on the queuing policy of the active event queues. Unless specified otherwise, the pending queue will implement the FIFO policy, as that trivially achieves the liveness property – at least one event is eventually scheduled from each event queue.

Thread pool scheduler. Task schedulers, such as the Fork/Join pool (Lea 2000) from the JDK, multiplex a set of tasks across a set of worker threads. It is useful to reuse the effort put into task schedulers when implementing a reactor scheduler. In the following, we show the `ExecutorScheduler`, which uses a JDK `Executor` to schedule reactor frames:

```
class ExecutorScheduler(val e: Executor)
extends Scheduler {
  def initSchedule(f: Frame) =
    f.schedulerState =
      new DefaultState with Runnable {
        def run() = execute(f) }
  def schedule(f: Frame) =
    executor.execute(f.schedulerState) }
```

Method `initSchedule`, called when the reactor starts, creates a default state with the JDK `Runnable` interface mixed in. Scheduler state is simultaneously a *task*, which calls the `execute` method from Figure 2. Method `schedule` then passes this task to the `Executor`, delegating the decision of when to run the reactor to a task-based scheduler.

Dedicated thread or process scheduler. In some cases, we want to give a specific reactor a higher priority by assigning it a dedicated thread or process. Here, the decision of when to run is delegated to the underlying OS.

Such a reactor need not process events in batches, and can simply flush all its event streams until they are empty, as shown in the following scheduler state implementation:

```
class DedicatedState extends State {
  def onBatchStart() {}
  def onBatchEvent(c: Connector) = true }
```

The `ThreadScheduler` uses an auxiliary method `loop`, which calls `execute` from Figure 2, and then waits inside a monitor until the reactor terminates or there is a pending event queue. The loop ends when the reactor terminates.

```
def loop(f: Frame) = do {
  execute(f)
  f.monitor.synchronized {
    while (!hasTerminated(f) && !hasPending(f))
      f.monitor.wait() }
} while (!hasTerminated(f))
```

When the reactor starts, the `ThreadScheduler` creates a `DedicatedState` object with a thread that calls `loop`. The thread is started in `schedule` the first time that the reactor is supposed to run, triggered by the `spawn` in Figure 1.

```
class ThreadScheduler extends Scheduler {
  def initSchedule(f: Frame) =
    f.schedulerState = new DedicatedState {
      val thread = new Thread {
        override def run() = loop(f) } }
  def schedule(f: Frame) =
    f.monitor.synchronized {
      if (!f.schedulerState.thread.isStarted)
        f.schedulerState.thread.start()
      f.monitor.notify() } }
```

The dedicated thread is subsequently notified when event delivery by `send` from Figure 3 prompts a call to `schedule`.

Piggyback scheduler. Normal programs are started by executing the `main` function on the main thread of the program. A reactor-based program has no notion of a main thread. It is therefore convenient to *piggyback* the existing main thread to one of the reactors in the program. This is the task of the following `PiggybackScheduler`:

```
class PiggybackScheduler extends Scheduler {
  def initSchedule(f: Frame) {}
  def schedule(f: Frame) =
    if (f.schedulerState == null) {
      f.schedulerState = new DedicatedState
      loop(f)
    } else f.monitor.synchronized {
      f.monitor.notify() } }
```

The first time `schedule` is called by the `spawn` method, the piggyback scheduler executes the event loop, thus blocking the current thread. Subsequently, `schedule` calls in the `send` method notify the thread that there are new events.

Fair scheduler. `Scheduler` implementations shown so far satisfy liveness, but they are not necessarily fair. An OS kernel or a task scheduler can satisfy fairness across a set of threads or tasks. However, neither has information about the number and age of events delivered to different reactors, and cannot give more time to reactors whose load is higher.

In the following, we show a scheduler that is fair according to the definition from Section 2.1. We use an event queue factory that assigns a timestamp to an event when it gets enqueued. The event queue itself respects the FIFO policy. Timestamp of the oldest event can be obtained by calling `headTime` on the queue. We define two helper methods that return a numeric priority of a connector and a reactor frame:

```
def cpriority(c: Connector[_]) =
  -1 * c.queue.headTime
def fpriority(f: Frame) =
  cpriority(f.pending.head)
```

The `pending` queue is a priority queue that sorts event queues using `cpriority`. The fair scheduler maintains another priority queue `tasks` for the set of activated reactor frames, based on `fpriority`. When a reactor is started, its event queue factory is replaced by a timestamping queue factory. The scheduler state uses the same connector as long as its priority is higher than the priority of the other activated frames, and other connectors of the current frame. The `schedule` method enqueues a frame to the `tasks` queue, and a separate thread dequeues and executes frames.

```
class FairScheduler extends Scheduler {
  private val tasks =
    new PriorityQueue[Frame](fpriority) {
  def newPendingQueue() =
    new PriorityQueue[Connector[_]](cpriority)
  def initSchedule(f: Frame) {
    f.queueFactory =
      new TimestampQueueFactory(f.queueFactory)
    f.schedulerState = new State {
      def onBatchStart() {}
      def onBatchEvent(c: Connector[_]) =
        cpriority(c) > fpriority(tasks.head) &&
        cpriority(c) > cpriority(f.pending.head)
    } }
  def schedule(f: Frame) = tasks.enqueue(f)
  startThread {
    while(true) execute(tasks.dequeue()) } }
```

This is a proof-of-concept implementation of a fair scheduler, which is neither scalable (because it is single-threaded) nor efficient (because $C = 1$). In practice, it is useful to relax fairness to some degree in order to achieve performance.

Timer scheduler. Real-time computations must be scheduled at regular intervals. The following `java.util.Timer`-based implementation periodically schedules a reactor.

```
class TimerScheduler(period: Long) {
  val timer = new java.util.Timer
  def initSchedule(f: Frame) {
    f.schedulerState = new DefaultState
    val task = new java.util.TimerTask {
      def run() {
        if (hasTerminated(f)) this.cancel()
        else execute(f) } }
    timer.schedule(task, period, period) }
  def schedule(f: Frame) {} }
```

The `initSchedule` method creates a new `TimerTask` that executes the frame, or cancels itself if the reactor terminated.

Resource scheduler. In some cases, a reactor must be scheduled only when a specific resource is available. A resource can be an external hardware sensor, an embedded coprocessor, or a GPU on commodity hardware.

```
class ResourceScheduler extends Scheduler {
  def initSchedule(f: Frame) =
    f.schedulerState = new DefaultState
  def schedule(f: Frame) =
    OS.requestResource(() => execute(f)) }
```

When `schedule` gets called, `ResourceScheduler` requests an OS resource and passes a callback that executes the frame once the resource becomes available.

5. Evaluation

We used standard benchmarks from the Savina suite (Imam and Sarkar 2014a) to test our scheduler performance (Prokopec 2016) against the Akka framework (Akka 2015). We used established evaluation methodologies (Georges et al. 2007) (Prokopec 2014). Benchmarks were done on a quad-core 2.8 GHz Intel i7-4900MQ processor with 32 GB of RAM, and results are shown in Figure 4.

Ping-Pong. In this benchmark, one (re)actor sends a preallocated *ping* message to another (re)actor, which then responds with a *pong* message. This is repeated N times. The benchmark evaluates how fast the scheduling system exchanges the context between two (re)actors when it is likely that they will be reactivated soon after deactivation. Our reactor implementation is around $1.6\times$ faster than Akka.

Streaming Ping-Pong. Akka frequently uses an alternative form of the Ping-Pong benchmark in which the first actor starts by sending W *ping* messages, instead of a single one. Whereas in the Ping-Pong benchmark each actor must wait for the reply before sending the next message, in Streaming Ping-Pong the two actors work on a sliding window of messages and do not have to yield control to the scheduler. Our reactor system is $1.3-1.4\times$ faster than Akka.

Thread Ring. Here, R (re)actors are arranged in a ring, and each waits for a message before sending it to the next (re)actor in the ring. Program ends after the message is forwarded N times. The benchmark tests context switching when it is unlikely that a (re)actor will be reactivated soon after deactivation. Depending on N, our system is in some cases as fast as Akka, and sometimes up to $1.2\times$ slower.

Counting Actor. A producer actor sends N numbers to a counter actor. The counter actor accumulates the sum of the numbers, and terminates. The benchmark is somewhat similar to Streaming Ping-Pong. Our implementation is around $3\times$ faster than Akka.

Fork Join (Throughput). A single (re)actor allocates K (re)actors that count incoming messages, and sends them N messages in a round-robin manner. The benchmark evaluates messaging throughput, and quality of batching messages. Our system is $2.5 - 3.0\times$ faster than Akka.

Fork Join (Creation). Benchmark creates N (re)actors, and sends a message to each of them. After a (re)actor receives a message, it terminates. This benchmark tests actor creation performance. Depending on N, our system is as fast as Akka, and sometimes up to $1.3\times$ faster.

Fibonacci. This benchmark computes Fibonacci numbers recursively, where each recursive call creates a (re)actor.

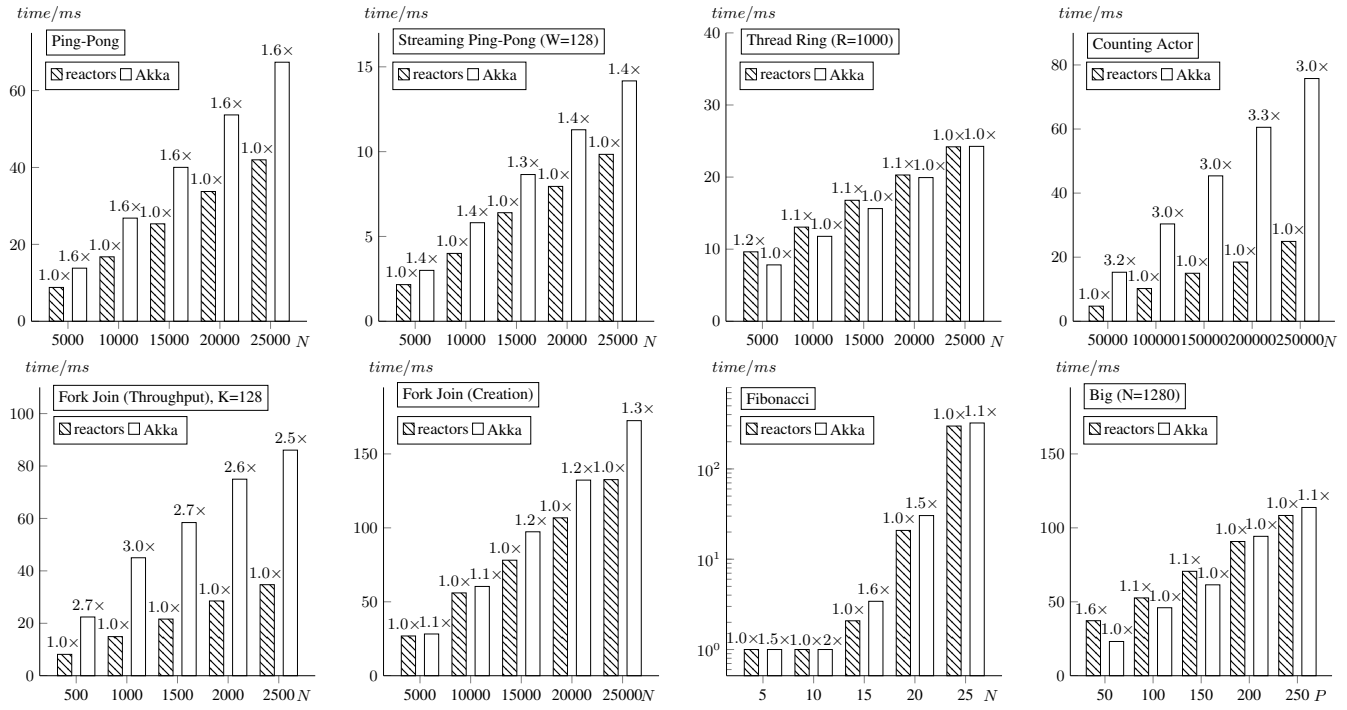

Figure 4. Running time on standard actor benchmarks (lower is better)

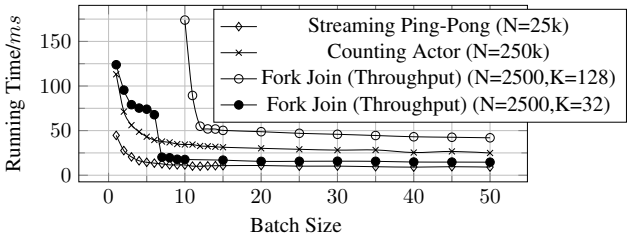

Figure 5. Dependence of running time on batch size for high message load benchmarks (lower is better)

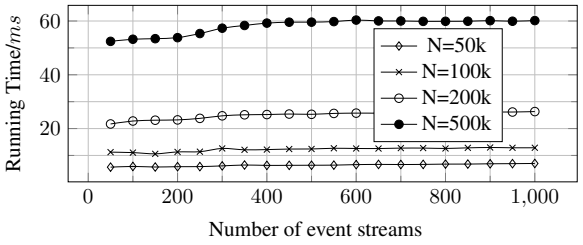

Figure 6. Roundabout Benchmark (Lower is Better)

This benchmark tests dynamic actor creation performance. Results are shown in logarithmic scale in Figure 4. Our system is $1.1 - 2.0\times$ faster than Akka, depending on N.

Big. This benchmark creates a large number of (re)actors N, each sending P pings to P randomly chosen (re)actors, awaiting a reply between each consecutive ping. The benchmark tests many-to-many message passing. Depending on P, our system is in some cases $1.1-1.6\times$ slower than Akka, and in some cases $1.1\times$ faster than Akka.

5.1 Effect of Batch Size on Performance

In benchmarks that have high average message count per actor, exchanging contexts between actors frequently can be detrimental. Amortizing these costs by handling multiple events in one batch greatly increases performance.

As argued in Section 2.1, batch size must be bound to ensure fairness – large batch sizes have a negative effect of delaying execution of other reactors. Batching must amortize context switch costs, but also prevent starvation.

In Figure 5, we show benchmarks where batch size affects the running time. In Streaming Ping-Pong and Counting Actor, the inflection point is around batch size 5, but performance converges above 40. We show Fork Join Throughput for different choices of the number of reactors K. We leave out out-of-scale points below batch size 10 for $K = 128$. For $K = 32$, we can see a steep jump around 7. Here, the batch size is just large enough to give inactive reactors sufficient time to fill their event queues. By the time a reactor is reactivated, it has sufficiently many pending events to benefit from batching. Based on these benchmarks, we keep the `BATCH_SIZE` constant from Section 4, at value 50.

5.2 Event Stream Scalability

We now show that the system scales with the number of event streams per reactor. In Figure 6, we show the *Roundabout benchmark*, in which the roundabout actor receives N messages on K event streams. The running time is almost constant when increasing the event streams count – gentle upward slope is a consequence of decreasing cache-locality.

6. Related Work

The actor programming model was proposed by Agha (Agha 1986). One of the most notable applications of the actor model is the Erlang programming language (Erlang 2015). On the JVM, Scala Actors followed the Erlang model (Haller and Odersky 2006), but since JVM does not have continuations, semantics were not equivalent to the Erlang-style `receive` statement. Akka is an actor-based framework (Akka 2015), which takes a step away from the Erlang model in that it supports only a single top-level `receive` statement. Kilim (Srinivasan and Mycroft 2008) is another JVM actor framework that takes a more sophisticated approach by exposing the `@pausable` annotation, used to mark and transform methods that potentially suspend. The Reactors.IO framework exposes event streams on which suspendable computations can be chained monadically or with a sequence of callbacks (Prokopec 2016) (Prokopec et al. 2014) (Prokopec and Odersky 2015).

Selector model is an actor model with multiple mailboxes (Imam and Sarkar 2014b). In this model, there are multiple mailboxes that the actor can programmatically activate. Although the abstract selector model allows a dynamic number of mailboxes, the implementation requires specifying the number of mailboxes before the selector starts.

Most actor schedulers are built on top of a task scheduler, such as the Fork/Join framework (Lea 2000). Depending on the task scheduler implementation, this approach ensures liveness. However, fairness, as defined in Section 2.1, is not necessarily ensured – giving all actors equal execution times can cause starvation when message-load is non-uniform.

Many frameworks use pluggability to defer some scheduling decisions to the client. For example, message scheduling in Akka (Akka 2015) uses the underlying task scheduler to assign equal execution chunks to actors, but this does not guarantee message handling fairness. It is the client's job to implement a fair *dispatcher* to customize the scheduling policy, or a custom event queue if additional capabilities such as persistence are necessary (Prokopec 2015). Parallel actor monitors (Scholliers et al. 2014) for the AmbientTalk language (Pinte et al. 2013) expose a user API that can optionally enable parallelism within an actor. Ensuring scalability is thus deferred to the client-side.

7. Conclusion

We described a scheduler algorithm for the reactor model. We showed that the scheduler satisfies safety properties and can guarantee liveness and fairness depending on the scheduling policy. Scheduling policies are pluggable – in addition to the default policies shown in Section 4, users can define their own custom policies. We have empirically shown that the scheduler is scalable and efficient by comparing our implementation against the industry-standard Akka framework, on the Savina actor benchmark suite.

An interesting area of future work is scheduling reactor programs on heterogeneous resources, such as GPUs. Achieving scalability and good performance in non-uniform computations is more challenging, and we believe that our pluggable scheduler infrastructure is well suited for this task.

References

G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. ISBN 0-262-01092-5.

Akka. Akka Documentation, 2015. http://akka.io/docs/.

Erlang. Erlang/OTP documentation, 2015. http://www.erlang.org/.

A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.*, 42(10):57–76, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297033.

R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer. ISBN 978-3-540-28845-9.

P. Haller and M. Odersky. Event-Based Programming without Inversion of Control. In *Proc. Joint Modular Languages Conference*, Springer LNCS, 2006.

S. M. Imam and V. Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. AGERE! '14. ACM, 2014a. ISBN 978-1-4503-2189-1. doi: 10.1145/2687357.2687368.

S. M. Imam and V. Sarkar. Selectors: Actors with Multiple Guarded Mailboxes. AGERE! '14, pages 1–14, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-2189-1. doi: 10.1145/2687357.2687360.

D. Lea. A Java Fork/Join Framework. JAVA '00. ACM, 2000. ISBN 1-58113-288-3. doi: 10.1145/337449.337465.

M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

K. Pinte, A. Lombide Carreton, E. Gonzalez Boix, and W. Meuter. Ambient Clouds: Reactive Asynchronous Collections for Mobile Ad Hoc Network Applications. Springer, 2013. ISBN 978-3-642-38540-7. doi: 10.1007/978-3-642-38541-4.

A. Prokopec. ScalaMeter, 2014. http://scalameter.github.io.

A. Prokopec. SnapQueue: Lock-Free Queue with Constant Time Snapshots. Scala '15, 2015. doi: 10.1145/2774975.2774976.

A. Prokopec. Reactors.IO Website, 2016. https://reactors.io.

A. Prokopec and M. Odersky. Isolates, Channels, and Event Streams for Composable Distributed Programming. Onward! 2015, pages 171–182, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3688-8. doi: 10.1145/2814228.2814245.

A. Prokopec, P. Haller, and M. Odersky. Containers and Aggregates, Mutators and Isolates for Reactive Programming. SCALA '14. ACM, 2014. doi: 10.1145/2637647.2637656.

C. Scholliers, E. Tanter, and W. De Meuter. Parallel Actor Monitors: Disentangling Task-level Parallelism from Data Partitioning in the Actor Model. *Sci. Comput. Program.*, 80:52–64, Feb. 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.03.011.

M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Jan. 2011.

S. Srinivasan and A. Mycroft. *Kilim: Isolation-Typed Actors for Java*, pages 104–128. Springer, Berlin, Heidelberg, 2008. doi: 10.1007/978-3-540-70592-5.