Pluggable Scheduling for the Reactor Programming Model

Aleksandar Prokopec(✉)

Oracle Labs, Zürich, Switzerland
aleksandar.prokopec@gmail.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× performance improvements on standard actor benchmarks.

1 Introduction

The recently proposed reactor model [3,26,31] 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 [31] dealt only with the programming model, but did not discuss the underlying implementation. The existence of multiple event streams within a single reactor poses a scheduling problem that differs from scheduling in 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 that belong 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 a perfect fit for every situation. A scheduler can be Pareto-optimal, meaning that there is no other scheduler that is equal

A. Ricci and P. Haller (Eds.): Programming with Actors, LNCS 10789, pp. 125–154, 2018.
https://doi.org/10.1007/978-3-030-00302-9_5

or better on all workloads. However, there may exist another scheduler that is better on one particular workload, but worse on some other workloads.

For example, in the Ping-Pong benchmark [15], a message is likely to arrive soon, 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 in most cases unlikely that a message will arrive soon. In these cases, users should be able to decide which scheduling policy is more appropriate for their workload, or implement adaptive schedulers that can dynamically adjust themselves to the conditions in the program.

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, a DSP or a temperature sensor reading), then the scheduler needs to negotiate the availability of the resource with the OS. A generic scheduler cannot do this, and this warrants a user-defined scheduling policy, which embeds such domain-specific knowledge into the scheduling mechanism.

This paper brings forth the following contributions:

- Detailed description and implementation of a scheduling algorithm for the reactor programming model (Sect. 3).
- A list of safety- and progress-related properties that a reactor scheduler must satisfy. We analyze the proposed scheduling algorithm, and show that it satisfies these properties under specific assumptions on the user-defined scheduling policy (Sects. 2.1 and 3.3).
- A pluggable mechanism for user-defined scheduling policies, which can embed application-specific knowledge (Sect. 4).
- An overview of optimization techniques used in our implementation, which was added to this extended version of the original paper [24] (Sect. 5).
- A performance comparison on the Savina benchmark suite [15] with the widely adopted Akka actor framework. We show that our reactor implementation outperforms Akka on 6 out of 8 benchmarks by a factor of $1.1 - 3.3\times$, and otherwise has comparable performance (Sect. 6).

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. 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 [14,38], and not the reactor scheduler. However, effective single-machine scheduling is likely a prerequisite for efficient distributed computations, since it ensures a better utilization of each machine.

This paper is an extended version of the previous work on pluggable scheduling for the reactor programming model [24]. In this version, we present an overview of optimization techniques that were used to improve the performance of our scheduler: explicit work stealing, lazy task scheduling, reanimation threads, message arrival speculation and actor class profiling (Sect. 5).

Code examples are written in Scala [20], 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.

We start by describing the reactor model in more detail in Sect. 2, and we then show the proposed scheduling algorithm in Sect. 3.

2 Reactor Model

The reactor programming model [31] is a generalization of the standard actor model [2,4]. There are three major differences between these 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 [31], 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, reactor model allows defining best-effort and reliable broadcasts, failure detectors [10,18] and CRDTs [34], and exposing them as reusable components, that can be either embedded into a reactor, or further composed into more complex components.

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 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

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

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]`, indicating that it delivers events of type `T`. In our case, `main.events` has the type `Events[String]`, because we declared a reactor of type `Reactor[String]`.

Unlike processes in the π-calculus [19], where a process can block until a message arrives on a channel, or join calculus [8], where a process can decide to block until another process sends a matching message, 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. Effectively, a reactor has multiple synchronous control flows. Concretely, 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 Stopped => 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" }
```

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 template, we 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, or its scheduling policy.

We claimed earlier that reactors generalize actors. To validate this, we encode an Akka-style [1] actor using a reactor from the Reactors.IO framework [3].

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` if the partial function is defined for it. 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 [31]. In a nutshell, the reactor model consists of the following components:

- **Defining and 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 and eventually handle them.
- **Sending events:** channels and the `!` operator, used to asynchronously send events to other reactors.
- **Modularising protocols:** the `open` method, used to create supplementary channels in the current reactor[2].

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 essential properties.

2.1 Properties of a Reactor Scheduler

We now explore important 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 execute and process some of its 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 data races that would otherwise result from simultaneously manipulating reactor state, and obviates the need for user-level synchronization. Importantly, serializability applies to events received on all event streams belonging to the same reactor – at most one handler across all event streams may be active at a time.

Fairness states that if an event is delivered to the reactor on some event stream, then the corresponding event handler is eventually invoked, unless the

[2] It was shown that encoding multiple protocols in the actor model using a single actor is possible, but made easier with custom protocol description languages [37]. Conversely, protocol modularisation can also be achieved by encapsulating groups of actors, each of which handles one aspect of the protocol. A custom architecture description language was made to facilitate this type of encapsulation [5]. One goal of the reactor model is to allow modularisation in the core model, without relying on another language layer.

event stream gets sealed by user code[3]. An important assumption that we make is that no reactor executes an infinite loop, i.e. the handling of every event consists of a finite number of steps.

Although fairness ensures that delivered events are eventually handled, a stronger guarantee is sometimes more useful. Whenever possible, a scheduler should avoid a scenario in which a set of events delivered to one event stream grows indefinitely[4]. This can, for example, occur in a multiple producer, single consumer setting. Fairness 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 *bounded delivery time 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 in the system. A reactor scheduling system is *scalable* if it meets the following conditions. First, event handling must not contend with 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 absolute execution time spent in scheduling must be negligible, or be amortized by the execution time of user code. This property is checked with an empirical evaluation.

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

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 fairness, bounded delivery time 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 Sect. 3, the proposed pluggable scheduling system enforces safety properties. Progress properties are good-to-have, but not essential for all programs.

[3] Users can do this explicitly in the reactor programming model, in which case the undelivered events are dropped. It is unclear to us how to achieve this using automatic GC, since there always exists a reference from the `onEvent` callback to the event stream object.

[4] This is not always possible – there exist programs in which the number of events grows over time. For example, if every reactor upon receiving an event sends out two events in response, then the overall number of messages in the program grows exponentially over time. However, if there exist an execution schedule in which the number of messages in the program at any given point is bounded, then the scheduler should use that execution schedule.

For such properties, the scheduling system establishes a well-defined foundation, and delegates the decision of fulfilling them to other components.

3 Scheduling Algorithm

In this section, we describe the proposed pluggable scheduling algorithm. We start by describing the internals of our reactor system implementation, and then show the algorithm itself. Finally, we prove that the algorithm satisfies serializability and fairness, and, under specific assumptions, can also achieve bounded delivery time fairness.

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 the event queue size. It can be called by any number of threads concurrently. Method `dequeue` 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 [13] – it can be called by at most a single thread at a time. When `dequeue` emits the event on the associated event stream, control transfers from the scheduler to the event handlers installed by the user code.

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

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. A *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. Method initSchedule is called once when the reactor is created, and schedule is called every time a reactor is activated. Method newPendingQueue creates a queue with a list of active connectors, and allows the scheduler to express a queuing policy.

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

Queue exposes standard queue operations enqueue and dequeue. Note that its implementation and queuing policy are different than that of an event queue. A *pending queue* stores *event queues*, and 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 Implementation

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 is placed 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 Fig. 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 New, 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 set a custom state object in the frame's schedulerState field. This is done in the call to initSchedule in line 13, and the default connector is allocated in line 14.

At this point, the frame is completely initialized and may begin execution. A call to activate in line 15 activates the frame. This method acquires the frame's lock in line 23, and checks if the frame is already active in line 24. If not, the active field is set to true in line 25. If the field active was set, the scheduler's schedule method is called in line 27. This indicates that there is a

```
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.lifecycle = New
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  f.main.channel }
20
21 def activate(f: Frame) {
22   var run = false
23   f.monitor.synchronized {
24     if (!f.active) {
25       f.active = true
26       run = true } }
27   if (run) f.scheduler.schedule(f) }
```

Fig. 1. Reactor creation

newly activated frame that should be scheduled on some thread. The scheduler should 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 Fig. 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 the auxiliary methods `checkNew` and `checkStopped` to treat newly created and stopped reactors differently. Method `checkNew` is called before event processing starts, and it atomically changes the state from `New` to `Running`. If the state changes to running, it means that this is the first time that the reactor was run, so the `checkNew` method needs to run the reactor constructor. It is important to run the constructor asynchronously, and not in the `spawn` method, to ensure non-blocking semantics. The constructor is run in line 17 with a call to the prototype's `create` method.

```
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 && f.lifecycle != Stopped) repeat = true
9        else f.active = false }
10     f.isolationCount.set(0)
11     if (repeat) f.scheduler.schedule(this) } }
12
13 def checkNew(f: Frame) {
14   var isNew = false
15   f.monitor.synchronized {
16     if (f.lifecycle == New) { f.lifecycle = Running; isNew = true } }
17   if (isNew) f.reactor = proto.create() }
18
19 def checkStopped(f: Frame, forced: Boolean) {
20   var stop = false
21   f.monitor.synchronized {
22     val isRunning = f.lifecycle == Running
23     val mustStop = f.pending.isEmpty && f.connectors.length == 0
24     if (isRunning && (forced || mustStop)) {
25       f.lifecycle = Stopped; stop = true } }
26   if (stop) f.system.release(name) }
27
28 def lifecycleAndProcessBatch(f: Frame) {
29   try { checkNew(f); processEvents(f) }
30   catch { case t: Throwable => checkStopped(f, true) }
31   finally checkStopped(f, false) }
32
33 def processEvents(f: Frame) {
34   f.schedulerState.onBatchStart(this)
35   val c = popPending(f); if (c != null) drain(c) }
36
37 def popPending(f: Frame): Connector[_] = f.monitor.synchronized {
38   if (f.pending.nonEmpty) f.pending.dequeue() else null }
39
40 @tailrec def drain(c: Connector[_]) {
41   val remaining = c.queue.dequeue()
42   if (f.schedulerState.onBatchEvent(c)) {
43     if (remaining > 0 && !c.isSealed) drain(c)
44     else {
45       val nc = popPending(f); if (nc != null) drain(nc) }
46   } else if (remaining > 0 && !c.isSealed)
47     f.monitor.synchronized { f.pending.enqueue(c) } }
```

Fig. 2. Reactor loop

The `checkStopped` method similarly checks for termination, and is called after handling the 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 `forced` argument 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 `Stopped`, the reactor name is released in line 26.

In practice, all these methods emit lifecycle events on the system event stream, but we omit them from Fig. 2 for brevity.

At this point, the reactor can start handling the delivered events. The method `lifecycleAndProcessBatch` first calls the method `processEvents`, which in line 34 notifies the scheduler that a batch of events is about to be handled. The `processEvents` method then calls `popPending` to dequeue a non-empty connector. If a reactor just started, it is likely that no events were yet delivered, and `popPending` returns `null`. In this case, `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 41. This releases an event on the corresponding event stream, and enters user code. After the event handlers process the event, `dequeue` returns the number of remaining events at the point in time when the event was removed. The `drain` method then asks the scheduler if it should continue executing events in line 42. 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 43, with the same connector. If the current event queue is empty, `drain` attempts to pop the next non-empty connector if there is one, and calls itself recursively in line 45. If the scheduler denies processing additional events, the `drain` method puts the non-empty event queue back to the pending queue in line 47.

Using the `onBatchEvent` method, 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, as explained in Sect. 6.1.

```
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     }
11   }
12   if (run) f.scheduler.schedule(this) }
```

Fig. 3. Event send

A reactor is also activated when an event is delivered on one of its event streams. This is done by the `send` method in Fig. 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 previously 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 12. The `execute` method from Fig. 2 is eventually invoked on some thread.

Note that the implementation of the `schedule` method must be synchronized, since multiple concurrent reactors can call `send` at the same time. We will show several implementations of the `schedule` method in Sect. 4.

3.3 Analysis of the Scheduling Algorithm

Having seen the scheduling algorithm, we state several claims about its properties. We prove that the algorithm is safe with respect to the serializability property. For space reasons, we skip other safety properties such as exactly-once delivery. We then prove fairness and bounded delivery time fairness, with specific assumptions about the `Scheduler` implementation.

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

Proof. No thread is initially running `execute`. The first call to `schedule` occurs in the `activate` method in Fig. 1, and the second `schedule` occurs in the `send` method in line 12 in Fig. 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 calls `schedule` until `execute` reaches line 11 in Fig. 2.

The `execute` method calls `schedule` in line 11 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 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[5]. □

Lemma 1 (Deactivation). *The reactor's `pending` queue never contains an event queue that is empty.*

Proof. We show this inductively – the 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 list. Events are only dequeued by the `drain` method from Fig. 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.*

Proof. An event is delivered to the event stream in line 3 of the `send` method shown in Fig. 3. If, in line 3, `enqueue` returns a size greater than 1, then there is another thread T for which `enqueue` previously returned 1. Between the point

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

in time t_0 when `enqueue` returned 1 for that other thread T, and the point in time t_1 when T puts the queue on the pending list, no other thread can drain that event queue, because that event queue is not yet on the pending list. By contradiction, assume that the event queue *is* on the pending list between t_0 and t_1. That would only be possible if, between t_0 and t_1, `enqueue` returned 1 for some other thread in line 3, which would imply that the event queue size became 0 between t_0 and t_1. That would be a contradiction, because the event queue can only be dequeued after being placed on the pending list, and, by Lemma 1, the event queue was not on the pending list at time t_0.

Now, consider the thread for which `enqueue` returns size 1 in line 3 of Fig. 3. That thread puts the event queue to the `pending` list after a finite number of steps. Next, consider the thread that calls `popPending`. If the event queue is non-empty when that thread subsequently calls `dequeue` in line 41, 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 (Fairness). *Assume that `schedule` eventually executes the specified reactor, and that every event queue added to the pending list gets removed after calling `dequeue` on the pending list 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 on the `pending` queue, or will be after a finite number of steps. By assumption, every reactor is eventually executed, and every event queue on the `pending` queue is eventually dequeued. For each such event queue, at least one event is handled. Consequently, every event is eventually handled. □

Achieving bounded delivery time fairness is deferred to the pluggable `Scheduler` object. Accordingly, the proof of the bounded delivery time fairness makes heavy assumptions on the `Scheduler` implementation.

Theorem 3 (Bounded delivery time 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 given assumptions, `dequeue` call in line 41 always returns the oldest event in the system. Therefore, scheduling is fair for the constant $C = 1$ in the bounded delivery time fairness definition from Sect. 2.1. □

4 Scheduling Policies

In this section, we go over several implementations of the `Scheduler` trait. There are several 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` objects expose the `onBatchStart` and `onBatchEvent` methods. The former is called when a reactor starts handling a batch of events, and the latter is called after handling each event of that batch. Most schedulers use some variant of the `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; return batch > 0 } }
```

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

Thread pool scheduler. Task schedulers, such as the Fork/Join pool [17] from the JDK, are designed to 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) }
```

The `initSchedule` method, called when the reactor starts, creates a default state with the JDK `Runnable` interface mixed in, so the scheduler state is simultaneously used as a *task*. When run by a task scheduler, this task object calls the `execute` method from Fig. 2. Method `schedule` 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 a 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 Fig. 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 (!hasStopped(f) && !hasPending(f)) f.monitor.wait()
  }
} while (!hasStopped(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 Fig. 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 whenever, during event delivery, the `schedule` method gets called from the `send` method from Fig. 3.

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` implementation:

```
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.

Scheduler with bounded delivery time fairness. The `Scheduler` implementations shown so far satisfy the fairness property, but they do not necessarily have bounded time delivery. An OS kernel or a task scheduler can satisfy bounded delivery time 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 Sect. 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. The timestamp of the oldest event can be obtained by calling `headTime` on the queue. We define a pair of helper methods that return the 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` list implementation is a priority queue that sorts the 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 BoundedDeliveryTimeFairScheduler 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 this requirement to some degree to achieve higher performance.

Timer scheduler. Real-time computations, such as interactive graphics rendering, must be scheduled at regular intervals. The following implementation is based on the `java.util.Timer`, and it periodically schedules reactor execution.

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

The `initSchedule` method creates a new `TimerTask` that periodically executes the frame, or cancels itself if the reactor has 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 general purpose GPU.

```
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 Scheduler Optimizations

The previous sections described the scheduling algorithm, with the emphasis on making it pluggable. The design is sufficient for making the scheduling generic, but we did not explain how we achieve high throughput. In this section, we go over several techniques that we used to improve the performance of the default scheduler implementation in the Reactors framework. The main goal in these techniques is to reduce the amount of time spent in the scheduler compared to the amount of time spent in application code.

5.1 Explicit Work Stealing

In a typical task-based work stealing scheduler [6], there is a set of worker threads, usually corresponding to the number of processors. Each worker thread has an associated work stealing queue. When the program creates a new concurrent task of execution in some thread, it places that task onto the work stealing queue. A task is removed from the queue either after the current task of the corresponding worker thread completes, or when another worker runs out of tasks on its own queue and *steals* the task from a queue of its peer.

There are several overheads associated with work stealing. First, a task object needs to be created and placed on the queue. Second, inactive worker threads need to be notified that there is work available for them. Third, a previously inactive worker thread must scan the work queues of other threads, and steal an available task. The overhead of having to wait can be reduced by having the active worker thread steal back the tasks it previously submitted as early as possible. Some task-based work stealing schedulers expose the API that allows the client code to explicitly start the work stealing process [17].

In our implementation, the worker thread dequeues the previously submitted tasks immediately after completing event handling. This allows executing a different reactor frame on the active worker thread before context switching the current frame. After the line 35 of the code in Figure 2, the worker calls a special method `stealSchedule`, shown in Fig. 4, which dequeues and runs frame tasks.

Piggybacking the worker thread can only have a single level of nesting. To ensure this, each worker thread keeps a `state` field that initially

```
1 def stealSchedule() {
2   if (state == STEAL_SCHEDULE) {
3     return
4   }
5   state = STEAL_SCHEDULE
6   try {
7     var loopsLeft = STEAL_SCHEDULE_COUNT
8     while (loopsLeft > 0) {
9       val executedSomething = pollWorkQueueAndRunFrame()
10      if (executedSomething) {
11        loopsLeft -= 1
12      } else {
13        loopsLeft = 0
14      }
15    }
16  } finally {
17    state = ASLEEP
18  }
19 }
```

Fig. 4. Worker thread piggybacking

has the value `AWAKE`, but switches to `STEAL_SCHEDULE` upon entering the `stealSchedule` method. Recursive calls to `stealSchedule` check this field and immediately return. The method then attempts to poll pending tasks up to `STEAL_SCHEDULE_COUNT` times.

This optimization is usually only beneficial if the time spent in the polling method `pollWorkQueueAndRunFrame` is short, since the scheduler spends useless cycles and slows the overall execution otherwise. For this reason, we only poll the local queue of the current worker thread in our implementation.

5.2 Lazy Task Scheduling

The optimization in Sect. 5.1 improves throughput by reducing the context switch pauses, but does not eliminate the cost of creating task objects. Compared to starting a thread, spawning a task is much more lightweight, but creating a task and pushing it onto a work stealing queue is still relatively expensive.

In this section, we describe an optimization that drastically reduces the number of tasks in some benchmarks. The optimization is based on the following observation: in most (re)actor programs, an execution of an event handler is relatively short (in fact, this is a recommendation in many frameworks, since monopolizing the worker threads with long event handlers can lead to message overflows). Furthermore, the longer the execution of an event handler is, the more likely it is that it will send messages to other reactors. This means that the time between sending two messages from a single event handler, and the time between sending a message and exiting from an event handler, are both usually short. Sending a message potentially results in scheduling a reactor frame, as shown in Fig. 3, which results in creating a task object. Consequently, postponing the task creation until the next `schedule` call or until the completion of the event handler usually does not have a negative performance impact.

```
1 val miniQueue = new AtomicReference[Frame](null)
2
3 @tailrec final def schedule(frame: Frame) {
4   val oldFrame = miniQueue.get
5   if (oldFrame eq null) {
6     if (!miniQueue.compareAndSet(oldFrame, frame)) {
7       schedule(frame)
8     }
9   } else {
10    if (!miniQueue.compareAndSet(oldFrame, null)) {
11      schedule(frame)
12    } else {
13      val task = createTaskFor(oldFrame)
14      pushToWorkQueue(task)
15    }
16  }
17 }
```

Fig. 5. Lazy scheduling of frame task objects

To exploit the previous observation, we augment each worker thread with an additional one-element queue, which we call a *mini-queue*, and use it to store reactor frames that were activated, but whose scheduling was lazily postponed. Initially, when a reactor frame gets execution time on some worker thread, the mini-queue is empty, i.e. set to `null`. When the reactor sends a message, and that message requires scheduling the reactor frame, the reactor frame is instead atomically placed on the mini-queue. If another reactor frame needs to be scheduled, the old reactor is first atomically removed, and then corresponding task object gets created and pushed to the work queue. This is shown in Fig. 5.

The code in Fig. 4 is similarly adjusted to check not only for pending work on the work stealing queue, but also flush the state of the mini-queue.

5.3 Reanimation Thread

The optimization from Sect. 5.2 trades latency for higher throughput. While in most cases, the decrease in latency is not observable, it is possible to write a program in which an event handler sends one message, but then does not return control to the scheduler for a long time. In these cases, an activated frame gets temporarily stuck on the mini-queue, and it cannot be stolen by other worker threads, since it is not on the work stealing queue.

To prevent this scenario, our implementation uses an additional non-worker thread, called a *reanimation thread*, that is usually in the sleep state, but occasionally wakes up and scans the mini-queues of the worker threads. When the reanimation thread finds a non-empty mini-queue, it atomically removes the frame and executes it. This is shown in Fig. 6.

When it is unlikely that there are any benefits from scanning the mini-queues, the reanimation thread should spend the least amount of CPU time possible. For this reason, after the reanimation thread inspects all the worker threads, it adjusts its sleep period to a new value that depends on the previous value, the

```
1 class ReanimationThread extends Thread {
2   setDaemon(true)
3
4   def run() {
5     var sleepPeriod = 0
6     while(true) {
7       Thread.sleep(sleepPeriod)
8       var count = 0
9       for (worker <- workers) {
10        val f = worker.miniQueue.get
11        if (f != null && worker.miniQueue.compareAndSet(f, null)) {
12          execute(f)
13          count += 1
14        }
15      }
16      sleepPeriod = adjust(sleepPeriod, count, workers.length)
17    }
18  }
19 }
```

Fig. 6. Reanimation thread implementation

number of removed frames and the total number of workers, as specified by the following equation:

$$adjust(period, count, workers) = 2 \cdot period \cdot \frac{workers - count}{workers} \tag{1}$$

In the above equation, if the *count* is low, then the new period value is increased by up to 2×. On the other hand, if the *count* is high, the period is rapidly decreased. The net effect is that when the reanimation thread helps almost all workers, the period is almost immediately reduced to a minimum value, and otherwise slowly decays towards the maximum value. When the benefit of flushing mini-queues is small, this approach reduces the overall CPU time spent by the reanimation thread. The new period is clamped between some minimum and maximum value, which is in our case between 1 ms and 200 ms.

5.4 Message Arrival Speculation

After a reactor executes an event handler, it normally undergoes a context switch and returns the control to the scheduler. In applications in which pairs of reactors exchange values, a message frequently arrives to the reactor shortly after it had begun its context switch. In these cases, it pays off to speculate on the arrival of the message and delay the context switch. This speculation is based on the bet that the message arrival time is lower than the time required for the context switch. To estimate the message arrival time, each reactor does sampling – a reactor occasionally spins before its context switch and counts the number of messages that arrived during spinning. The sampling frequency must be low to avoid slowing down the program. After accumulating a set of samples, the reactor decides whether to regularly spin before the context switch, or not.

Importantly, message arrival speculation is dynamic – it is only applied if the scheduler detects that it is likely to improve performance. If during the sampling period, a reactor does not notice any benefits from delaying the context switch, then it will not apply speculative spinning.

Concretely, a reactor decides to speculate that a message will arrive as follows. Define the time period δ as the ration between the context switch delay d and the total time required to do the context switch c (which is estimated experimentally, by sending messages to a reactor with an empty event handler). The reactor decides to delay context switches by a relative delay δ if and only if the percentage of arrivals $\hat{p}$ during sampling satisfies:

$$\delta \leq \hat{p} \tag{2}$$

The intuition behind this is that delaying the context switch potentially slows down the program. To make up for this slowdown, the relative delay must be smaller than the percentage of cases in which it accurately predicted that a message will arrive, avoiding the full context switch. The reactor must pick the optimal value for δ, so it collects a sample for a range of values δ_i and $\hat{p}_i$, and picks the one for which the following expression is minimized:

$$\delta_i - \hat{p}_i \tag{3}$$

After deciding on the context switch delay, the reactor continues sampling the delay benefits, and repetitively revisits the speculation decision after collecting new sample sets. This way, if message arrival pattern changes during the execution, the reactor adjust the context switch delay.

In Fig. 7, we show the performance of two benchmarks from the Savina suite [15] for which message arrival speculation is particularly effective. In the *Fork*

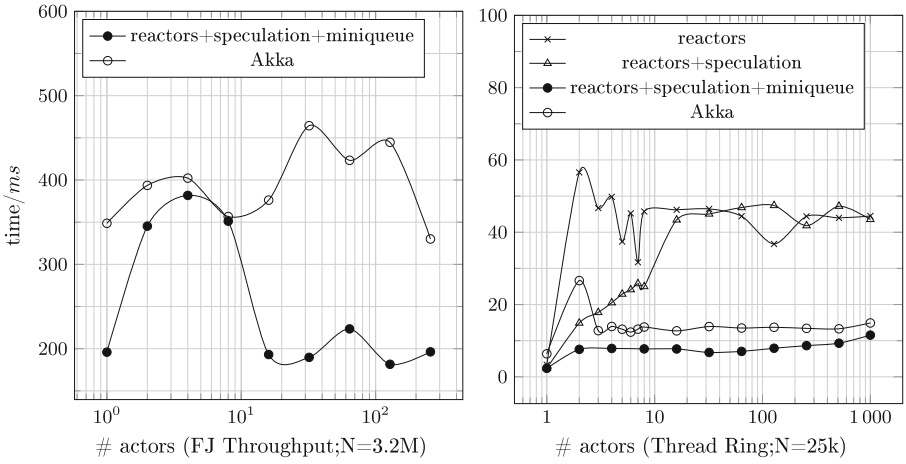

Fig. 7. Impact of the parallelism level on different benchmarks with message arrival speculation enabled (running time vs. number of reactors)

Join Throughput benchmark, a reactor allocates K other reactors, and sends each of them a message in a round-robin manner. The performance improvement on the *Fork Join Throughput* benchmark is high when K is above 8, which is the number of processors on the test machine. The reason is that when there is a higher number of reactors, speculative spinning allows messages to pile up at some of the reactors, so when the reactor gets scheduled, the batch of messages it can process is larger. For K below 8, performance becomes close to that of the Akka framework. The *Thread Ring* benchmark allocates R reactors that pass a token around in a ring pattern. Here, the situation is reversed – it only makes sense to spin and wait for the next message if each reactor can be pinned to a processor (which is true if the ring size R is small). Otherwise, spinning delays the assignment of a processor to inactive reactors, and slows down the program overall. We show the performance of Reactors compared to Akka, when speculative spinning is disabled, when it is enabled, and when we additionally apply the lazy task scheduling optimization (mini-queue).

For other benchmarks, message arrival speculation does not yield any benefits. Nevertheless, it is important to ensure that the sampling overhead does not compromise overall performance, so the sampling frequency φ is initially kept at a low value, in our case 0.2%. The downside of doing this is that it takes a long time to collect a sample set when speculation *is* beneficial. To retain the best of both worlds, a reactor adapts the sampling frequency according to the belief that arrival speculation helps. When and if a reactor notices an arrival of a message (but before it has gathered a full sample needed for its speculation decision), it adaptively increases the sampling frequency. A reactor is allowed to do this, since sampling is itself a context switch delay – a higher sampling frequency is therefore not detrimental when context switch delays help.

In Fig. 8, we show some of the benchmarks on which speculation does not help, namely *Thread Ring* (with $R \gg \#processors$), *Streaming Ping-Pong*, *Big* and *Fibonacci*. We plot the running time of the benchmark with respect to the initial sampling frequency φ_0. Based on these benchmarks, we decided to keep φ_0 at the value 0.2%.

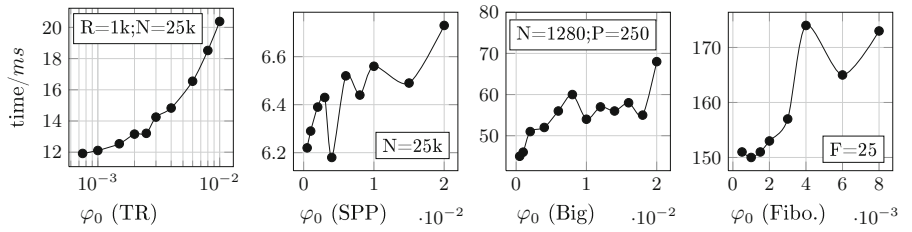

Fig. 8. Impact of different sampling frequencies on performance for benchmarks that do not benefit from speculation (running time vs. sampling frequency)

A theoretical model of the message arrival speculation optimization was described recently, along with a more in-depth performance analysis [25]. For more information, we refer the readers to related work.

5.5 Class-Based Reactor Profiling

Speculation described in Sect. 5.4 works well only if the lifetime of a reactor is sufficiently long to gather a sample that estimates the benefits of delaying context switches. In many applications, the average lifetime of a reactor is shorter than that, so reactors do not manage to effectively estimate speculation benefits. This is true even if a reactor adaptively adjusts the sampling frequency as described in Sect. 5.4. To accomodate these cases, our implementation periodically profiles the stable sampling frequency with respect to the class of the reactor. This information is stored in a per-class histogram, and used as the initial sampling frequency when a reactor gets created. This way, the information about the ideal sampling rate gets shared among reactors of the same type, and applications with short reactor lifetimes can also benefit from speculation. In our implementation, the lifetime of the sampling frequency profile is specific to a single execution of a program.

When profiling numeric values such as the sampling frequency, the more recent information must be given a higher importance. The intuition is that the applications are going through phases, and the optimal values may change over time. We use the following expression to update the sampling frequency:

$$record(oldRate, newRate) = 0.8 \cdot oldRate + 0.2 \cdot newRate \tag{4}$$

By using the expression (4), the more recent profiling information quickly starts dominating, and the old value tends to decay and become less important over time. Every profiled value is continuously interpolated in this way.

6 Evaluation

We used standard actor benchmarks from the Savina suite [15] to test the performance of our scheduler (code online [3]) against the industry-standard Akka framework [1]. We used established evaluation methodologies [9,22]. Benchmarks were done on a quad-core 2.8 GHz Intel i7-4900MQ processor with 32 GB of RAM, and the results are shown in Fig. 9.

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. This benchmark is designed to test how fast the scheduling system exchanges the context between two (re)actors, when it is likely that they each will be reactivated soon after deactivation. Our reactor implementation is around 1.6× faster when compared to Akka.

Streaming Ping-Pong. The Akka project [1] often uses an alternative form of the Ping-Pong benchmark in which the first actor starts by sending W *ping*

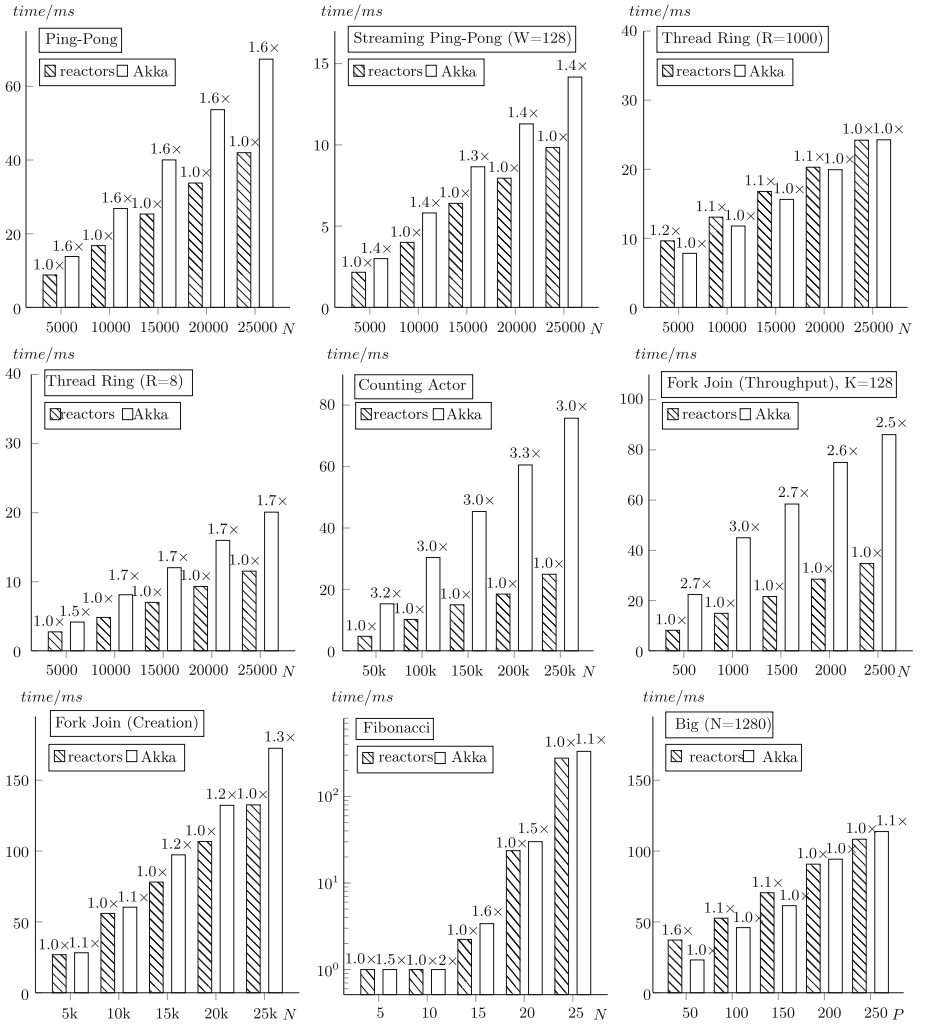

Fig. 9. Running time on standard actor benchmarks (lower is better)

messages, instead of a single one. Whereas in the Ping-Pong benchmark each actor waits for the reply before sending the next message, in Streaming Ping-Pong, the two actors keep a sliding window of messages and usually do not 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. The program ends after the message is forwarded N times. When R is much larger than the processor count, the benchmark tests context switching when it is unlikely that a (re)actor will be reactivated soon after deactivation. When $R = 1000$, depending

on N, our system is in some cases as fast as Akka, and sometimes up to 1.2× slower. For $R = 8$, our system is up to 1.7× faster compared to Akka.

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, but also evaluates the efficiency of allocating messages. Reactors have typed channels that can specialize on the message type, and can avoid boxing. In our implementation, we rely on the type specialization optimization in Scala [7] that avoids boxing primitive types such as integers. For this (scheduler-unrelated) reason, our implementation is around 3× 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× faster.

Fibonacci. This benchmark computes Fibonacci numbers recursively, where each recursive call creates a (re)actor that sends the result to its parent. For sizes $N < 2$, leaf actors send 1 immediately after creation. This benchmark tests dynamic actor creation performance. Results are shown in logarithmic scale in Fig. 9. Our system is $1.1 - 2.0\times$ faster than Akka, depending on N.

Big. This benchmark creates a large set of (re)actors N, each sending P pings to P randomly chosen (re)actors, awaiting a reply for each 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× faster than Akka.

6.1 Effect of Batch Size on Performance

In benchmarks that have high average message count per actor, exchanging contexts between actors frequently can be detrimental. Each context switch requires relatively expensive state checks from Fig. 2, and a call to the `schedule` method. Amortizing these costs by handling multiple events in one scheduling batch greatly increases performance.

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

In Fig. 10, we show a selection of benchmarks where batch size affects the benchmark running time. In Streaming Ping-Pong and Counting Actor, the inflection point is around batch size 5, but performance converges above 40. We show the Fork Join Throughput benchmark for different choices of the number of reactors K. We leave out out-of-scale points below batch size 10 for $K = 128$.

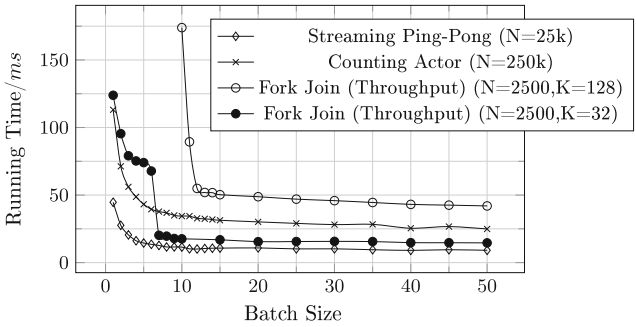

Fig. 10. Dependence of the running time on the batch size for benchmarks with a high message load (lower is better)

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 Sect. 4, at value 50.

6.2 Event Stream Scalability

The last thing to show is that the system scales with the number of event streams per reactor. In Fig. 11, we show our custom *Roundabout benchmark*, in which the roundabout actor receives N messages on K different event streams. The running time remains almost constant while increasing the number of event streams – the gentle upward slope is a consequence of decreasing cache-locality.

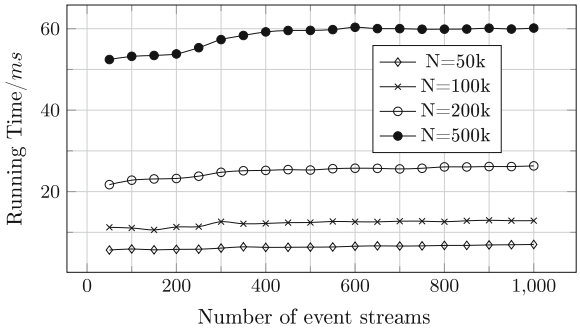

Fig. 11. Roundabout Benchmark (Lower is Better)

7 Related Work

The actor programming model was proposed by Agha [4], and has gone through many variants since then. One of the most notable applications of the actor model

is the Erlang programming language [2]. On the JVM, Scala Actors attempted to mimic the Erlang model [11] – since JVM does not have continuations, semantics of the Erlang-style `receive` statement could not be completely imitated. Akka is a widely adopted actor-based framework [1], which takes a step away from the Erlang model in that it supports only a single top-level `receive` statement. Kilim [36] 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 as first-class objects. The advantage of first-class streams is that suspendable computations can be chained as a sequence of callbacks [3,28,31].

Selector model is an actor model variant with multiple mailboxes [15]. In this model, there are multiple guarded mailboxes that the actor can programmatically activate or deactivate. Although the abstract selector model allows a dynamic number of mailboxes, the current selector implementation requires specifying the number of mailboxes before the selector starts [16]. The scheduling algorithm is based on the multi-level queue scheduling [35], used to separate mailboxes into priority levels. This separation does not necessarily ensure fairness, so authors mention LRU round-robin scheduling as necessary future work.

The Kilim framework introduced the concept of *scheduler hopping*. Here, an actor can programmatically change the scheduling policy during the lifetime of an actor. Neither Akka, nor Reactors.IO allow changing the policy after actor creation. We believe that it is easy to add scheduler hopping to Reactors.IO.

Most actor schedulers are built on top of a task scheduler, such as the Fork/Join framework [17]. Depending on the task scheduler implementation, this approach ensures fairness. However, bounded delivery time fairness, as defined in Sect. 2.1, is not necessarily ensured – giving all actors equal execution times can cause starvation when the message-load is non-uniform. We have not yet found programs where this is problematic, and the prebundled schedulers in Reactors are not fair. On the other hand, it is common to have schedulers that extend the Fork/Join task scheduler with domain-specific knowledge, and this was done for actor systems [1], asynchronous programming frameworks [12], streaming frameworks [29], and for data-parallel collections [27,30].

Many frameworks use pluggability to defer some scheduling decisions to the client. For example, message scheduling in Akka [1] uses the underlying task scheduler to assign equal execution chunks to actors, but this does not guarantee message handling bounded delivery time fairness. It is the client's job to implement a fair *dispatcher* if that is necessary. Aside from custom dispatchers, both Akka and Reactors allow users to inject their own message queue implementations. This is advantageous in use-cases such as message persistence, for which more efficient queue data structures exist [23,32]. Parallel actor monitors [33] for the AmbientTalk language [21] expose a user API that can optionally enable parallelism within an actor. Ensuring scalability is thus deferred to the client-side. The Mesos cluster runtime [14] has a very thin scheduler based on *resource offers*, which does not even guarantee fairness. This may be an indication that programs where fairness is an issue are in practice rare, and can be

dealt on a case-by-case basis. At the same time, pluggable schedulers are useful, as they allow clients to deal with pathological edge cases when they occur.

8 Conclusion

We described a scheduler algorithm for the reactor programming model, and presented its implementation. We showed that the scheduler satisfies safety properties such as handling at most one message in a reactor at a time, and also guarantees fairness and bounded delivery time fairness with specific guarantees from the scheduling policy. Scheduling policies are pluggable – in addition to the default policies shown in Sect. 4, users can define their own custom policies. We empirically showed 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 that additionally use heterogeneous resources of the host system, such as GPUs, DSPs and external sensors. Achieving scalability and good performance in such non-uniform computations is more challenging, but also potentially more rewarding. We believe that our pluggable scheduler infrastructure is well suited for this task.

References

1. Akka Documentation (2015). http://akka.io/docs/
2. Erlang/OTP documentation (2015). http://www.erlang.org/
3. Reactors, IO Website (2016). https://reactors.io
4. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
5. Astley, M., Agha, G.A.: Customization and composition of distributed objects: middleware abstractions for policy management. In: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 1998/FSE-6, pp. 1–9. ACM, New York (1998). http://doi.acm.org/10.1145/288195.288206
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999). http://doi.acm.org/10.1145/324133.324234
7. Dragos, I., Odersky, M.: Compiling generics through user-directed type specialization. In: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, pp. 42–47. ACM, New York (2009). http://doi.acm.org/10.1145/1565824.1565830
8. Fournet, C., Gonthier, G.: The Join Calculus: a Language for Distributed Mobile Programming, September 2000. https://www.microsoft.com/en-us/research/publication/join-calculus-language-distributed-mobile-programming/
9. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. SIGPLAN Not. **42**(10), 57–76 (2007). http://doi.acm.org/10.1145/1297105.1297033

10. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-642-15260-3
11. Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Lightfoot, D.E., Szyperski, C. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 4–22. Springer, Heidelberg (2006). https://doi.org/10.1007/11860990_2
12. Haller, P., Prokopec, A., Miller, H., Klang, V., Kuhn, R., Jovanovic, V.: Scala Improvement Proposal: Futures and Promises (SIP-14) (2012). http://docs.scala-lang.org/sips/pending/futures-promises.html
13. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
14. Hindman, B., et al.: Mesos: a platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI 2011, pp. 295–308. USENIX Association, Berkeley (2011). http://dl.acm.org/citation.cfm?id=1972457.1972488
15. Imam, S.M., Sarkar, V.: Savina - an actor benchmark suite: enabling empirical evaluation of actor libraries. In: Proceedings of the 4th International Workshop on Programming Based on Actors Agents and Decentralized Control, AGERE! 2014, pp. 67–80. ACM, New York (2014). http://doi.acm.org/10.1145/2687357.2687368
16. Imam, S.M., Sarkar, V.: Selectors: actors with multiple guarded mailboxes. In: Proceedings of the 4th International Workshop on Programming Based on Actors Agents and Decentralized Control, AGERE! 2014, pp. 1–14. ACM, New York (2014). http://doi.acm.org/10.1145/2687357.2687360
17. Lea, D.: A Java fork/Join framework. In: Proceedings of the ACM 2000 Conference on Java Grande, JAVA 2000, pp. 36–43. ACM, New York (2000). http://doi.acm.org/10.1145/337449.337465
18. Lynch, N.A.: Distributed Algorithms. MK Publishers Inc., San Francisco (1996)
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. I. Inf. Comput. **100**(1), 1–40 (1992). https://doi.org/10.1016/0890-5401(92)90008-4
20. Odersky, M., al.: An Overview of the Scala Programming Language. Technical report IC/2004/64, EPFL Lausanne, Switzerland (2004)
21. Pinte, K., Lombide Carreton, A., Gonzalez Boix, E., De Meuter, W.: Ambient clouds: reactive asynchronous collections for mobile ad hoc network applications. In: Dowling, J., Taïani, F. (eds.) DAIS 2013. LNCS, vol. 7891, pp. 85–98. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38541-4_7
22. Prokopec, A.: ScalaMeter Website (2014). http://scalameter.github.io
23. Prokopec, A.: Snapqueue: lock-free queue with constant time snapshots. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala 2015, Portland, OR, USA, 15–17 June 2015, pp. 1–12 (2015). http://doi.acm.org/10.1145/2774975.2774976
24. Prokopec, A.: Pluggable scheduling for the reactor programming model. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, pp. 41–50. ACM, New York (2016). http://doi.acm.org/10.1145/3001886.3001891
25. Prokopec, A.: Accelerating by idling: how speculative delays improve performance of message-oriented systems. In: Rivera, F.F., Pena, T.F., Cabaleiro, J.C. (eds.) Euro-Par 2017. LNCS, vol. 10417, pp. 177–191. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64203-1_13
26. Prokopec, A.: Encoding the building blocks of communication. In: accepted at Onward 2017, to appear (2017)

27. Prokopec, A., Bagwell, P., Rompf, T., Odersky, M.: A generic parallel collection framework. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6853, pp. 136–147. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23397-5_14
28. Prokopec, A., Haller, P., Odersky, M.: Containers and aggregates, mutators and isolates for reactive programming. In: Fifth Annual Scala Workshop, SCALA 2014, pp. 51–61. ACM (2014). http://doi.acm.org/10.1145/2637647.2637656
29. Prokopec, A., Miller, H., Schlatter, T., Haller, P., Odersky, M.: FlowPools: a lock-free deterministic concurrent dataflow abstraction. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 158–173. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37658-0_11
30. Prokopec, A., Odersky, M.: Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. In: Caşcaval, C., Montesinos, P. (eds.) LCPC 2013. LNCS, vol. 8664, pp. 55–86. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09967-5_4
31. Prokopec, A., Odersky, M.: Isolates, channels, and event streams for composable distributed programming. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), Onward! 2015, pp. 171–182. ACM, New York (2015). http://doi.acm.org/10.1145/2814228.2814245
32. Prokopec, A., Odersky, M.: Conc-trees for functional and parallel programming. In: Shen, X., Mueller, F., Tuck, J. (eds.) LCPC 2015. LNCS, vol. 9519, pp. 254–268. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29778-1_16
33. Scholliers, C., Tanter, E., De Meuter, W.: Parallel actor monitors: disentangling task-level parallelism from data partitioning in the actor model. Sci. Comput. Program. **80**, 52–64 (2014). Feb
34. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A Comprehensive Study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, January 2011. https://hal.inria.fr/inria-00555588
35. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 8th edn. Wiley Publishing (2008)
36. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_6
37. Sturman, D.C., Agha, G.A.: A protocol description language for customizing failure semantics. In: Proceedings of IEEE 13th Symposium on Reliable Distributed Systems, pp. 148–157, October 1994
38. Verma, A., Pedrosa, L., Korupolu, M.R., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with borg. In: Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France (2015)