Molecule: Using Monadic and Streaming I/O to Compose Process Networks on the JVM

Sébastien Bocq and Koen Daenen
Bell Labs, Alcatel-Lucent
Antwerp, Belgium

November 10th 2012

Abstract

Molecule is a domain specific language library embedded in Scala for easing the creation of scalable and modular concurrent applications on the JVM. Concurrent applications are modeled as parallel process networks that exchange information over mobile and type-safe messaging interfaces.

In this paper, we present a concurrent programming environment that combines functional and imperative programming. Using a *monad*, we structure the sequential or parallel coordination of user-level threads, without JVM modifications or compiler support. Our mobile channel interfaces expose reusable and parallelizable higher-order functions, as if they were *streams* in a lazily evaluated functional programming language. The support for graceful termination of entire process networks is simplified by integrating channel poisoning with monadic exceptions and *resource control*. Our runtime and system-level interfaces leverage message batching and a novel flow parallel scheduler to limit expensive context switches in multicore environments. We illustrate the expressiveness and performance benefits on a 24-core AMD Opteron machine with three classical examples: a thread ring, a genuine prime sieve and a chameneos-redux.

Contents

1 Introduction

The design of concurrent systems has been studied for several decades already [Kahn, 1974] and is getting increasing attention now that multicore and cloud computing platforms are a commodity. Many implementations depend on specialized languages and runtimes to make efficient use of parallelism on multicore architectures. More recently, there have been several efforts [Karmani et al., 2009] to provide efficient solutions on mainstream JVMs. This is to leverage a mature and widely available execution platform, preserve interoperability with a vast ecosystem of libraries and recycle the expertise gained by a large community of programmers. A number of important challenges faced in the design of these systems are the following:

- How to leverage parallelism and shield programmers from data race issues?
- How to coordinate interactions with non-blocking I/O interfaces, which are asynchronous and callback-driven, in a robust, scalable and composable manner?
- How to handle asynchronous termination, both in normal and exceptional cases?
- How to schedule efficiently concurrent tasks in multicore environment?

To address the first point and ease reasoning about concurrent applications, a well established pattern is to structure these applications as networks of asynchronous and sequential activities that do not share state and interact exclusively over messaging interfaces. The second and third points are related to each other. The main issue with programming directly against callback interfaces is that the continuations [Clinger et al., 1988] capturing the different control flows of a program have to be threaded explicitly by the programmer through his program at the detriment of expressiveness. It violates encapsulation of the execution state as offered by threads [von Behren et al., 2003]. Furthermore, it forces developers to propagate termination information *explicitly* though the control flows of an application, both in normal and exceptional cases, which is intrusive and error prone. Existing work [Srinivasan, 2010; Rompf et al., 2009] can restore the illusion of a lightweight threaded programming model in asynchronous settings in such a way that it integrates well with the native exception handling mechanism present on mainstream JVMs using compile-time continuation-passing style (CPS) transformations. However, a programmer is still responsible for propagating termination information explicitly through the data flows embodied by a network of asynchronous activities, which is again intrusive and error prone. Unfortunately, compile-time transformations are themselves not trivial to maintain compared to pure library approaches. They are also difficult to extend because they interact in non obvious manner with the host platform threading model and language features such as exceptions and recursion. To gain a better understanding and control over termination, expressiveness and scheduling issues, this paper presents instead a pure library approach that leverages Scala's [Odersky et al., 2011] support for functional programming on the JVM.

In this paper, we describe Molecule, a concurrent programming library that runs on unmodified JVMs. Molecule combines a process-oriented programming model [Sampson, 2010], a domain specific embedded language (DSEL) [Hudak, 1998] and a run-time system. We exploited the higher extensibility of a pure library approach to offer a programming environment that combines the following advantages:

- *An application-level threading model with support for sequential execution, parallel execution and space-efficient tail calls.* Molecule's user-level threading programming model [Anderson et al., 1992] implements a continuation-passing form of *monadic I/O* [Peyton Jones and Wadler, 1993] at application-level to encapsulate lightweight process interactions over non-blocking messaging interfaces in a simple and composable manner.

These user-level threads are not limited to sequential interactions and can interact safely in parallel over multiple type-safe messaging interfaces within a process in a non-blocking manner. In contrast to native JVM threads, they permit also space-efficient tail calls within the monadic domain. Our users can thus express sophisticated protocol state machines as in Erlang [Virding et al., 1996] without the burden of using a code generator [Banker et al., 1998].

- *Control structures for automating the graceful termination of parallel process networks.* Graceful termination is a well known pattern proposed by Welch [1989] to prevent deadlocks and resource leaks in process networks. However, the mechanism relies on the explicit propagation of a termination signal, which is tedious and error prone. As an extension, our monadic threading model offers resource control structures coupled with exception handling to *seamlessly* propagate termination information over the messaging interfaces of a process that terminates, without explicit support from the application.

- *Word-at-a-time and streaming communication primitives.* Compared to most concurrent programming frameworks, our library also addresses the Von Neumann Bottleneck [Backus, 1978] by not restricting its messaging interfaces to *word-at-a-time* primitives only. Its type-safe communication interfaces can be transformed lazily using higher-order functions, like *streams* in functional programming languages. Supporting streaming primitives on communication interfaces not only improves the expressiveness but also permits the efficient processing of message batches.

- *A flow parallel scheduler that exploits the structure of process networks to optimize parallel resources usage.* The execution of parallel process networks is known to suffer inherently from a massive amount of context switches [Ritson et al., 2009]. To alleviate this issue, we devised a scheduler that exploits our high-level abstractions to eliminate expensive and unnecessary kernel-level context switches dynamically, whenever the data flow is purely sequential.

The structure of the paper is as follows. Section 2 describes how Molecule models concurrent systems after process networks. Section 3 provides a detailed description of monadic user-level threads along with our extensions for parallel interactions and exceptions. Section 4 describes our extensible channel interfaces with their support for streaming primitives. Section 5 introduces the resulting programming model and our flow parallel scheduler. In Section 6, we revisit several classical concurrent programming examples to illustrate the expressiveness of our programming model. We augment the description with microbenchmarks to show the positive impact of our runtime optimizations in multicore environments. Section 7 discusses the related work and Section 8 concludes.

A monad [Wadler, 1995] is a well known abstraction for computations that has been extensively studied in the context of Haskell [Peyton Jones and Hughes, 1999], a non-strict and pure functional programming language. We do not assume any prior knowledge of Haskell but we require the reader to be familiar with Scala 2.8 and its support for functional programming.

2 The Molecule Model

Most concurrent programming frameworks aimed at real-world usage diverge from the seminal mathematical models that inspired their design. For instance, many approaches claim to be derived from the Actor model [Hewitt et al., 1973; Agha, 1986] but they differ more or less importantly with it [Karmani et al., 2009]. Molecule's model cannot be strictly categorized either. To avoid confusion with existing models, we begin with its description and the terminology used throughout this paper in relation to previous work (Section 2.1). Finally, we list the conventions clients of our library must follow to avoid the data race issues stemming from hosting a concurrent DSEL in an impure language like Scala (Section 2.2).

2.1 Model and Terminology

In Molecule, process networks are composed from three abstractions: *processes*, *channel interfaces* and *communication channels*. A process embodies a *coordination* and *computing* activity [Gelernter and Carriero, 1992] and interacts with its external environment exclusively by manipulating channel interfaces. A channel interface is unidirectional — it has either input or output modality. A communication channel, or a channel for short, represents the set of physical means allocated for the transport of information between its interfaces.

For performance reasons, a process can input/output a list of messages, which we call a *segment*, from/to a channel interface in a single I/O operation. In our model, channel interfaces are *mobile* in the sense that they can be referenced as first-class values inside messages. One could say we differ from the original π-calculus [Milner, 1999] in that we distinguish between channels and their interfaces, and that the channel themselves are not mobile. Yet, we share the same intent, which is to ease the expression of *reconfigurable systems* that reorganize their network dynamically by exchanging channel information.

A process can also install one or more transformation functions, which we call *transformers*, on a channel interface, which can be thought as a *stream* in functional programming languages. For example, a transformed interface may map a function repeatedly to the segments that pass through it during I/O operations. A transformation may be statful or not. It may also be temporary. For example, prepending a message on a channel interface transforms this interface into a buffered one until the next I/O operation.

In addition to being dynamically reconfigurable, a process network is also *elastic*. It grows dynamically when processes create new channel interfaces or new processes, or install a new transformation function on a channel interface. Channel interfaces are responsible for allocating resources for a given channel on a physical medium. A process network shrinks when a process terminates, when a transformation ends, or when channel interfaces are terminated.

A process may terminate a channel interface using a *signal* that indicates the reason for termination. This is called *poisoning* because the signal spreads like poison through the network. The behavior of an interface poisoned by a process depends on its modality. When an output interface is poisoned, all the messages it may have buffered will be processed but any additional message output to it will be poisoned. A message that is poisoned will never be delivered. Therefore, to ensure the propagation of termination information, we require that a poisoned message propagates the signal to all the channel interfaces it may reference. We call this mechanism *message poisoning*. If an input interface is poisoned with a certain signal, then any message buffered on the interface is poisoned with the same signal and no new message will be delivered on that interface. In the remaining of this paper, we will say that an input is poisoned and an output is closed to distinguish between both behaviours. Interfaces poisoned by a process must clear the resources they allocated on a channel and propagate the poison signal to the underlying channel. The subsequent behavior is channel specific. For example, a channel may decide or not to propagate the poison signal to all its other interfaces. Additionally, interfaces must poison themselves with an error signal if they detect a failure on the underlying channel. In both cases, the poisoning behavior is inverted when interfaces are poisoned from the channel side. In all cases, whenever an interface is poisoned, the corresponding signal is always returned together with the result of the last I/O operation.

Output channel interfaces can be poisoned synchronously during the emission of the last segment via an out-of-band signal token. Save for the last output operation, processes synchronize with a channel by waiting for the result of an I/O operation. We call it *channel-driven scheduling*. By holding back the result of an I/O operation, channels can enforce ad-hoc flow control schemes, independently from how processes are implemented. This permits for example to prevent the risks of unbounded growth of message queues present in purely asynchronous messaging systems.

For example, a *cooperative channel* is a point-to-point channel interconnecting two processes within the same network. It exposes, therefore, both an input and an output channel interface. The channel-driven scheduling policy of this kind of channel enforces that, irrespective of the order in which processes interact, there is never more than one input or output operation pending at a time. Note that, as explained in the previous paragraph, a process can proceed immediately after outputting its *last* segment, regardless of whether an input operation is pending or not on the same channel[1]. Because of this subtle distinction, cooperative channels are not strictly equivalent to *synchronous channel* or *rendezvous channel* used in implementations of communicating sequential processes (CSP) [Hoare, 1978].

2.2 Restrictions on Side Effects

Processes and channel transformation functions may be executed in parallel. Therefore, the following restrictions must be applied to prevent data races:

1. Processes are isolated, i.e. they must not share references to mutable state.

2. Higher-order streaming primitives must accept only pure functions.

In Scala, the first requirement can be enforced by leveraging existing work on a uniqueness type plugin [Haller and Odersky, 2010]. Enforcing the second one requires a type and effect system in Scala, which is work in progress at the time of this writing [Rytz et al., 2012]. In their absence, we rely on the good will of the clients of our library to follow these rules as conventions. We will assume that these conventions are strictly followed in the remainder of this paper.

3 Monadic User-Level Threads

This section describes the design of our lightweight and extensible threading programming model. Then, we introduce two extensions not available with native JVM threads: interleaved parallelism and user-level exceptions with resource control. The support for batching, higher-order streaming primitives and poisoning will be introduced in Section 4.

3.1 Monadic Style

In many concurrent programming frameworks, non-blocking programs are written in so-called *callback style*. In this style, non-blocking methods accept a callback handler as parameter and return immediately while the result is computed asynchronously. In Scala, a callback is simply a function of type `A => Unit`, where `A` is an abstract parameter type. The callback function captures a *one-shot and effectful continuation closure* — it is called once and captures the rest of an effectful computation from a given point in a process. To shield application developers from callback interfaces, Molecule introduces a touch of laziness by encapsulating callback-driven

[1]Note that we apply the argument that a synchronous handshake between intermediate parts of a distributed system is not a substitute for end-to-end delivery mechanisms [Saltzer et al., 1984]. Reliable end-to-end delivery cannot be guaranteed in absence of end-to-end feedback.

effects behind the `IO` class shown in Figure 1. This class is essentially combining a mutable-state monad and a specialized form of continuation monad, called `Responder` in Scala's standard library.

```
trait UThread {
  def submit(task: => Unit):Unit
  def platform:Platform
}

final class IO[+A](
  val ask:(UThread, A => Unit) => Unit
) {
  def bind[B](react:A => IO[B]):IO[B] =
    new IO((t, k) => ask(t, a => react(a).ask(t, k)))

  def >>\[B](react:A => IO[B]):IO[B] = bind(react)
  def >>[B](next: => IO[B]):IO[B] = bind(_ => next)

  def flatMap[B](f:A => IO[B]):IO[B] = bind(f)
  def map[B](f:A => B):IO[B] =
    new IO((t, k) => ask(t, a => k(f(a))))
}

object IO {
  def apply[A](a: => A):IO[A] = new IO((t, k) => k(a))
}
```

Figure 1: The IO class.

In monadic style, non-blocking methods implement the `ask` function, which takes a user-level thread `UThread` and a continuation of type `A => Unit` as argument, but return it wrapped in an object of type `IO[A]`, commonly called *action*. A user-level thread is responsible for mapping the execution of the continuation tasks of a lightweight process to a common multitasking system abstracted by the `Platform` class. This assumes that the application of a continuation to its result is rescheduled by an asynchronous action as a task to its accompanying user-level thread. Below is a short example illustrating the encapsulation of an asynchronous function called `fooAsync` into an asynchronous action `foo`:

```
def fooAsync(a:A, k:B => Unit):Unit = ...

def foo(a:A):IO[B] =
  new IO[B]((t, k) =>
    fooAsync(a, b => t.submit(k(b)))
  )
```

The `submit` methods captures the application of the continuation as a task using a call-by-name parameter. The ability to control this mapping is valuable. For example, a platform may schedule continuation tasks over a dedicated thread pool or over a single threaded executor whose thread local storage is set to an OpenGL rendering context.

In contrast to callback methods, an action defers the execution of a continuation until its `ask` function is invoked. We chose to call this function "ask" to emphasize the demand-driven and asynchronous execution, i.e. one *asks* for the result of an action, and executes it, by applying its `ask` function to a user-level thread and a continuation. A crucial benefit of introducing laziness via this extra level of indirection, compared to programming against callback interfaces directly, is that actions can be composed sequentially into bigger actions in a type-safe and modular manner using the monadic binding combinator `bind`, or the equivalent ≫\ operator[2]. The `bind` operator permits to chain the next action as a function of the result of the previous one; we call this function *reaction*. For example, assuming the abstract types `A` and `B` are strings, to bind the result of the action `foo` to another call to `foo`, one can write:

```
foo("a") >>\ { s => foo(s) }
```

instead of:

```
new IO[String]((t, k) =>
  fooAsync("a", s => t.submit(
    fooAsync(s, ss => t.submit(k(ss)))
)))
```

The ≫ operator is another common monadic operator used to sequence two actions that are not related by the intermediate result. The method `flatMap` and `map` are methods required to use monadic actions in Scala's for-comprehensions. The factory method, commonly called *unit* or *return* in monadic terms, is implemented by the `apply` method of the companion object. It creates a new action that results in the value of its call-by-name parameter.

It can be easily proven, by applying equational reasoning rules in Scala, that the `IO` class satisfies the three monad laws. These laws ensure that the monadic combinators constrain the execution of effectful actions in a robust and intuitive manner, independently from the *evaluation* order or from the side effects *executed* within the `ask` function. Intuitively, this can be deduced by observing that the composition of monadic actions does not produce any side effect — only their execution does.

Although the user-level thread interface bears some similarities with the executor service interface found in the standard Java library, it differs in that it must obey the two additional requirements listed in the paragraphs below.

Space-efficient recursion (R1). Since we are building a system where processes interact with the external world only through non-blocking channel interfaces, the side effects of a process can only be the byproducts of executing continuations, which will themselves trigger other continuations, and so on. Therefore, the execution of a process is recursive and the stack of the underlying native thread must be unwound after executing each continuation task to prevent risks of stack overflows.

Sequential task execution (R2). We require that a user-level thread executes each task it gets submitted *sequentially* and that the effects of one task are *visible* to the next task in a *happens-before* relationship [Manson et al., 2005]. This eases reasoning about the behavior of the implementation but also guarantees that mutable data structures can be manipulated safely from within a process, even if a process interacts over multiple channel interfaces simultaneously.

Space-Efficient Tail Calls Given the fact that asynchronous actions resubmit their continuation to the user-level thread, thanks to R1, the invocation of an asynchronous action in tail call position in a bind expression does not create additional stack or heap usage. This property can be demonstrated easily using equational reasoning in Scala, writing the justification for each step in the right hand column. The proof depends on two intermediate lemmas.

[2]We chose this operator over the `>>=` operator commonly used in Haskell because of special precedence rules applied to operators with the `=` symbol in Scala.

Lemma 1 (L1). *An action that has no other effect than passing control to another action m is equivalent to that action.*

$$\texttt{new}\ \mathit{IO((t,\ k)\ =>m.ask(t,\ k))} \equiv m$$

Proof: By construction, to ask for the result of the left-hand side of the equation is equivalent to ask for the result of m.

Lemma 2 (L2). *To ask for the result of the action created by binding the action m to a function f is equivalent to apply f to the result of m, and then to ask the result of the action thus created.*

$$\mathit{(m \gg\!\backslash\ f).ask(t,\ k)} \equiv \mathit{m.ask(t,\ a\ =>\ f(a).ask(t,\ k))}$$

Proof:

```
(m >>\ f).ask(t, k)
  ≡ new IO((t, k) =>                         bind
      m.ask(t, a => f(a).ask(t, k))
    ).ask(t, k)
  ≡ m.ask(t, a => f(a).ask(t, k))            (L1)
```

Theorem. *The execution of an action bound in a tail position does not increase stack or heap usage.*

Proof: Given requirement R1, the stack is unwound each time an action is executed. Given L2, the continuation `k` passed to an action resulting from a bind operation is the same as the one passed to the action bound in tail position. This does not create new objects or reference older objects on the heap.

Examples Thanks to their support for space-efficient tail calls, our monadic threads permit the definition of recursive control structures or finite state machines as easily as in Erlang [Wadler, 1998]: just have one method returning an action for each state, with state transition represented by a bind to an action in tail call position. For example, without any additional library support, we can easily create a space-efficient looping structure that repeats an action a given number of times before returning control with the unit value:

```
def repeat[A](n:Int)(action:IO[A]):IO[Unit] =
  if (n == 0) IO(())
  else action >> repeat(n - 1)(action)
```

We can contrast the native and the monadic thread models on a simple example where a process writes the first string it receives from its input to its output in lower case, and then writes the ten subsequent strings it receives in upper case. According to our model, we implement each interaction of a process with the environment as a read or write I/O operation on a channel interface. As such a process can be written as a main function that takes a number of input and output interfaces as arguments, and returns the final result of the process. The natively threaded version of this process that uses blocking I/O interfaces is shown on Figure 2. The non-blocking version, based on our lightweight threading mechanism to interact over monadic I/O interfaces, is shown in Figure 3. Since we have defined the `map` and `flatMap` methods, we can take advantage of for-comprehensions to sequence actions.

```
trait Input[+A] {
  def read():A
}
trait Output[-A] {
  def write(a:A):Unit
}

def main(
  in:Input[String], out:Output[String]
):Unit = {
  val s = in.read();
  out.write(s.toLowerCase);
  for (i <- 0 until 10) {
    val s = in.read();
    out.write(s.toUpperCase);
  }
}
```

Figure 2: Example using a native thread.

```
trait Input[+A] {
  def read():IO[A]
}
trait Output[-A] {
  def write(a:A):IO[Unit]
}

def main(
  in:Input[String], out:Output[String]
):IO[Unit] = for {
  s <- in.read()
  _ <- out.write(s.toLowerCase)
  _ <- repeat(10) {
    for {
      s <- in.read()
      _ <- out.write(s.toUpperCase)
    } yield ()
  }
} yield ()
```

Figure 3: Example using a monadic thread.

Notwithstanding some minor syntactic noise created by the for-comprehension and the `IO` type used to denote effects, the non-blocking version reads the same way as a natively threaded implementation. In both cases, the thread executing the process is threaded transparently through sequential statements that appear on consecutive lines in the program.

In some cases, the syntax of the for-comprehension may appear a bit clumsy and it is not uncommon for seasoned functional programmers [Peyton Jones et al., 1996] to rely directly on monadic binding (≫\ ) and sequence (≫) combinators like this:

```
in.read() >>\ {out.write(_.toLowerCase)} >>
repeat(10) {
  in.read() >>\ {out.write(_.toUpperCase)}
}
```

Whichever style of monadic expression one chooses, it represents a substantial improvement in clarity and modularity compared to what can be achieved by programming directly against callback interfaces. For completeness, we must however mention two potential causes of mistakes. One comes from how Scala's for-comprehensions are desugarized and the other from the lack of a type and effect system.

A tail call within a for-comprehensions is not space-efficient. An infinite loop written using a for-comprehension:

```
def forever[A](action:IO[A]):IO[Nothing] =
  for (_ <- action; _ <- forever(action)) yield ()
```

is expanded by the compiler to:

```
def forever[A](action:IO[A]):IO[Nothing] =
  action.flatMap(_ => forever(action).map(_ => ()))
```

The return type `IO[Nothing]` indicates that the method does not return normally. Since the call to `forever` is followed by a `map`, it is not in tail call position and each iteration will accumulate a continuation until the program exhausts the heap space. Therefore, one must take care that tail calls appear only as argument to the `bind` combinator, for example by binding the tail call after a for-comprehension.

Actions on consecutive lines must be separated by the sequence combinator. If one forgets the monadic sequence operator ($\gg$) between actions appearing on consecutive lines, the code will compile, but only the action on the last line will be executed. Such error could be caught at compile-time using a type directed compiler plugin aware that `IO` objects do not induce side effects until they are executed.

3.2 Interleaved Parallelism

The `bind` operator ensures that an action is executed in reaction to a previous action. A process may also interact over multiple channels and execute several unrelated actions on each channel simultaneously before joining the final results. The execution of two unrelated actions can be coordinated by defining an additional `par` combinator on the `IO` class, as shown in Figure 4.

```
def par[B](other:IO[B]):IO[(A, B)] =
  new IO[(A, B)]({(t, k) =>
    var first:Option[Any] = None

    val ka:A => Unit = {a =>
      first match {
        case None       => first = Some(a)
        case Some(b:B)  => k((a, b))
      }
    }
    val kb:B => Unit = {...}

    this.ask(t, ka)
    other.ask(t, kb)
  })
```

Figure 4: The parallel interleaving combinator.

The `par` combinator creates a new action that executes, in one task, the current action and the action passed as parameter, and then collects each result in a pair. The two actions will have their continuation tasks scheduled by channel interactions independently from each other, in a non-deterministic manner. Since a user-level thread executes continuation tasks sequentially (R2), we call it *interleaved parallelism*. This is the reason why we can safely store the result of the first action using a mutable variable until the second result becomes available.

Note that this combinator shares the same algebraic properties as the *fork* operator discussed by Jones and Hudak [1993]. It is associative and it is also commutative only *if* both actions executed are themselves commutative, this last property being left as a proof obligation on the programmer.

3.3 Exceptions, Resource Control and Graceful Termination

Being able to signal the end of communication is required in order to support finite communication between processes. For normal execution it is sufficient to support a close operation on an output interface and let a channel propagate a signal downstream. However, a receiver may stop communicating as well, because of a network or hardware failure, a bug, or other interruptions. Although this is not a substitute for transactions, a mechanism to inform a sender process to stop using resources because its messages cannot be delivered has been proposed by Hilderink [2005]. In this scheme, an output interface can be invalidated downstream with a poison signal. Subsequent interactions on that interface raise an exception that can be caught inside a process using an exception handler scoped around that interaction. If the process cannot handle the exception, it is responsible for propagating it by poisoning its other interfaces in the exception handler before terminating. However, propagating explicitly poisoning information within exception handlers to ensure graceful termination is intrusive and very brittle given that spurious errors often occur where you least expect them. To address these issues, we augmented our monadic threads with a resource control mechanism that poisons automatically channel interfaces acquired within a block guarded by an exception handler. The library can then wrap the main action of a process within an exception handler such that any channel interface it acquired dynamically (including the initial ones) is automatically poisoned once the process terminates. Since we share the same signal type for both user-level exceptions and poison signals, the termination information will be *seamlessly propagated* through the channels registered as resource to the current context. The scheme encompasses normal termination, user-level exceptions raised by poisoned channels and Java runtime exceptions.

```scala
trait Signal
case object EOS extends Signal
case class JThrowable(t:Throwable) extends Signal

trait Resource {
  def shutdown(signal:Signal):Unit
}

class Context extends Resource {
  def add(r:Resource):Unit = ...
  def remove(r:Resource):Unit = ...
  def shutdown(s:Signal):Unit = ...
}

class UThreadContext(
  val uthread:UThread,
  val context:Context,
  val raise:Signal => Unit,
  val fatal:Signal => Unit
) extends UThread {

  def submit(task: => Unit):Unit =
    uthread.submit {
      try { task } catch {
        case t => fatal(JThrowable(t))
      }
    }

  def platform:Platform = uthread.platform

  def askInNewContext[A](
    ask:(UThreadContext, A => Unit) => Unit,
    k:A => Unit,
    handle:Signal => Unit):Unit = {

    val newContext = new Context()

    def cleanup(signal:Signal) = {
      newContext.shutdown(signal)
    }

    val newUThread = new UThreadContext(
      uthread, newContext,
      {signal =>             // (3) non-fatal exception
        cleanup(signal); handle(signal)},
      {signal =>             // (2) fatal exception
        cleanup(signal); fatal(signal)}
    )

    ask(newUThread, {a =>    // (1) normal termination
      cleanup(EOS); k(a)})
  }
}
```

Figure 5: The user-level thread context.

The UThreadContext class shown in Figure 5 augments a user-level thread with exception and resource control. Its constructor takes as parameter the user-level thread to augment, a Context object used to track managed resources and two continuations used to handle exceptional control flows. Channel interfaces, or any other resource inheriting from the Resource trait, can be registered to the local resource control by calling the add method on the Context object referenced by the user-level thread. This implementation distinguishes between non-fatal and fatal exceptions and assumes that the corresponding raise and fatal continuations call the shutdown method of the Context object to propagate the signal to the resources it manages. Java run-time exceptions caught inside the submit method during the execution of a task are passed as a JThrowable signal to the fatal continuation. The raise continuation is called by a poisoned channel interface to raise its signal as a user-level exception when a process attempts to read or write to it.
For example, processes can raise an exception using the following method[3]:

```
def raise(signal:Signal):IO[Nothing] =
  new IO[Nothing]((t, _) => t.raise(signal))
```

The askInNewContext method scopes the execution of a specific action to a new context and manages the different control flows such that the contract is respected. It takes as paraleter the ask function of the action to execute, a continuation k on which to pass the result of the action if there is no exception, and another continuation handle that is invoked if a non-fatal signal is raised during the execution of the action. Whichever way the action scoped by this method terminates (see comments (1), (2), (3) in Figure 5), all the resources it registered to the new context during its execution are automatically poisoned by the cleanup method. If the action terminates normally (1), its resources are poisoned with a standard end-of-stream (EOS) signal before the continuation k is applied to its result. If an exception occurs (2 and 3), resources are poisoned with the corresponding signal. The subsequent behavior depends on whether it is a fatal exception or not. A Java runtime exception, which is raised as a fatal exception (2), cannot be intercepted within the process where it occurs and ultimately causes the termination of that process by poisoning all its channels. Yet, since its channels have been poisoned, the signal will be raised as a non-fatal exception (3) within a neighbour process that attempts to interact on one of these channels. Non-fatal exceptions are passed to the handle continuation. This hook is used in conjunction with the orCatch method defined on the IO monad we refactored below to let application developers implement fault-tolerance actions for signals defined by a partial function:

[3]We could resubmit the continuation to the user-level thread in synchronous actions as well to prevent exhausting the stack in case the given action is expected to be called many times successively. However, we don't find it necessary as this situation does not occur in practice.

```scala
final class IO[+A](
  val ask:(UThreadContext, A => Unit) => Unit
) {...
  def orCatch[B >: A](
    handler:PartialFunction[Signal, IO[B]]
  ):IO[B] = new IO({(t, k) =>
      t.askInNewContext(ask, k, {signal =>
        if (handler.isDefinedAt(signal))
          // continue in the parent context
          handler(signal).ask(t, k)
        else
          // raise in the parent context
          t.raise(signal)
      })
    })
}
```

Since `orCatch` scopes resource control and exception handling to the action they target, exception handlers can be safely nested within each other. Depending on whether a signal handler is defined or not for an exception signal, the signal will be handled by a user-defined action in the parent context or propagated up the chain of exception handlers.

Custom coordination methods defined by application developers can take advantage of this resource control. For example, the control structure we defined previously to repeat an action can be rewritten like this:

```scala
def managed[A](action:IO[A]):IO[A] =
  action.orCatch { case signal => raise(signal) }

def repeat[A](n:Int)(action:IO[A]):IO[Unit] =
  if (n == 0) IO(())
  else managed { action } >> repeat(n - 1)(action)
```

The call to `managed` creates an execution context for the action by wrapping it inside a user-level exception handler. This ensures that all the resources acquired by the repeated action are automatically cleared up after each iteration.

Note that the implementation of `par` we saw previously is made slightly more complex by the introduction of user-level exceptions. For instance, it must invoke `ask` in a new context to handle potential signals raised by interleaved actions and ensure that the control is returned only when both actions have terminated, successfully or not. Note also the remarkable encapsulation properties of monadic threads. We introduced more sophisticated control flows, including state, without impacting the expressiveness of any of the programs shown earlier as examples.

4 Communication Channels

This section describes Molecule's channel interfaces. For extensibility and reusability concerns, the design of our channel interfaces has been split over three distinct layers, each one with well defined responsibilities.

System-level channel interfaces encapsulate side effects. Developers implement these interfaces to create new I/O channels that convert messages into real-world side effects and back, independently from our user-level threading model (Section 4.1). *Stream channel interfaces* wrap system-level channels into channels that expose reusable streaming primitives implemented by the library (Section 4.2). *Process-level interfaces* wrap stream channels into the monadic I/O interfaces introduced in the previous section. These are registered as resources to the user-level thread context of a process and manipulated conveniently from actions, using either word-at-a-time or streaming primitives (Section 4.3). Thanks to this separation of concerns, new system-level interfaces, whether they are blocking or not, can be integrated easily in Molecule. This is also facilitated by the hybrid imperative-functional nature of Scala, which lets developers access directly low-level interfaces exposed by the JVM.

4.1 System-Level Channel Interfaces

In Molecule, all communication effects are encapsulated within the minimalist system-level channel interfaces listed in Figure 6. Stream generators, cooperative channels, timer channels and other I/O channels offered by the library or defined by the end user must implement one or both of these interfaces.

```
trait SysIChan[+A] {
  def read(k:(Seg[A], SysIChan[A]) => Unit):Unit
  def poison(signal:Signal):Unit
}

trait SysOChan[-A] {
  def write(data:Seg[A], sigOpt:Option[Signal],
            k:SysOChan[A] => Unit):Unit
  def close(signal:Signal):Unit
}

case class NilChan(signal:Signal) extends
  SysIChan[Nothing] with SysOChan[Any]
{
  def read(...):Unit = sys.error("Nil:" + signal)
  def write(...):Unit = sys.error("Nil:" + signal)
  def poison(signal:Signal):Unit = {}
  def close(signal:Signal):Unit = {}
}
```

Figure 6: System-Level Channel Interfaces.

These interfaces are modeled after a standard recursive pattern in Haskell to lift side effects inside immutable settings. In this pattern, any modification brought to a stream returns a reference to the channel used in subsequent interactions, also called a *seed* [Coutts et al., 2007]. Here, we rely on the continuation `k` to return the seed lazily and we expose an extra method to poison channels asynchronously. A system-level input channel `SysIChan` encapsulates batches of side effects produced by a communication interface inside an immutable stream segment abstracted by the class `Seg`. A system-level output channel `SysOChan` abstracts the reverse operation: it consumes immutable segments to produce side effects. In addition, system-level output channels support synchronous poisoning by allowing an optional signal to be piggy backed with the last message sent. Clients of these channel interfaces can detect termination, and extract the poisoning signal indicating the cause of termination, by matching the next seed channel against the invalidated channel `NilChan`.

Since we require that channel interfaces are non-blocking, the only proof of obligation for the developer that wishes to implement a channel that must perform blocking I/O is to store a reference to the continuation and reschedule blocking tasks in an external thread managed behind the channel. The continuation, suspended on a read or write operation, can then be invoked back later by the thread after it has completed the corresponding blocking I/O operation. For example, the one shot timer channel shown in Figure 7 relies on an external executor for scheduling an event after a given delay and holds the read continuation in a private variable `kopt` until the event occurs.

```
object TimerChan {

  import java.util.concurrent._
  val executor:ScheduledExecutorService = ...

  def after(delay:Long, unit:TimeUnit):SysIChan[Unit] =
    new SysIChan[Unit] with Runnable {

      type K = (Seg[Unit], IChan[Unit]) => Unit
      val sf = executor.schedule(this, delay, unit)
      var kopt:Option[K] = None
      var ready = false

      def read(k:K):Unit = synchronized {
        if (ready) k(Seg(()), NilChan(EOS))
        else kopt = Some(k)
      }

      def run() = synchronized {
        kopt match {
          case Some(k) =>
            kopt = None; k(Seg(()), NilChan(EOS))
          case None   => ready = true
        }
      }

      def poison(signal:Signal) = synchronized {
        sf.cancel(false)
        kopt foreach {k => k(NilSeg, NilChan(signal))}
      }
    }
}
```

Figure 7: A one shot timer channel.

4.2 Stream Channel Interfaces

Stream channel interfaces (Figure 8) are implemented by the library. They provide reusable streaming primitives on top of system-level channels, which are automatically lifted to stream channels using implicit conversions. The role of the `Message` context bound can be ignored for the moment; it will be described further down this section.

```
trait IChan[+A] {
  def complexity:Int
  def read(t:UThread,
           k:(Seg[A], IChan[A]) => Unit):Unit
  def add[B:Message](
    transformer:IChan[A] => IChan[B]):IChan[B]
  def poison(signal:Signal):Unit

  def ::[B >: A :Message](b:B):IChan[B] =
    add(IBufferT(b))
  def map[B:Message](f:A => B):IChan[B] =
    add(IMapperT(f))
  def filter(p:A => Boolean)
            (implicit ma:Message[A]):IChan[A] =
    add(IFilterT(p))
  ...
}

trait OChan[-A] {
  def write(t:UThread, data:Seg[A], sigOpt:Option[Signal],
            k:OChan[A] => Unit):Unit
  def poison(signal:Signal):Unit
  def add[B:Message](
    transformer:OChan[A] => OChan[B]):OChan[B]

  def map[B:Message](f:B => A):OChan[B] =
    add(OMapperT(f))
  ...
}
```

Figure 8: Stream Channel Interfaces.

Stream input channels have the following features and responsibilities:

- They reschedule the application of continuations invoked by system-level channels as a task to the user-level thread passed as argument to their `read` or `write` method (R1).
- They stack higher-order transformations passed to the `add` method on the underlying channel. The library offers various stream transformation functions, we call stream transformers, for filtering, mapping functions to, parsing, grouping or splitting streams (in this case, the remainder of a stream is returned via another continuation), and so on. For convenience, the trait exposes several methods named after classical list operations. Some transformations may not be permanent. For example, the prepend operator `::` defined on this trait adds a transformer that returns a new channel whose read operation returns a segment containing the prepended message together with the original channel.
- They *parallelize* stream transformations above a *complexity cutoff threshold* (CCT) configured by the user. This threshold limits the number of synchronous transformations stacked on a channel, which is tracked by the `complexity` method. When the threshold is exceeded, the block of transformations previously stacked on the channel will be parallelized transparently by the library and the complexity of the channel resulting from the last transformation is set back to 0. The threshold should not be taken too large to avoid exhaustion of the stack space. However, there is some room for tuning this value since the stack size left to a new transformation is given by the formula $StackSize = StackSize_{Max} - complexity$,

where $StackSize_{Max}$ is the maximum stack size of native threads, which is usually quite high.

- They *split segments* when their size exceeds a library wide *segment size threshold* (SST) configured by the end user. The remaining part of a segment is prepended to the next seed channel. Using large segments may improve the throughput of applications while using shorter ones helps to preserve the reactiveness of the system.

- They may *fuse* stream transformations *dynamically* to eliminate intermediate segment creation [Wadler, 1988]. A similar technique described by Coutts et al. [2007] is used as a *compile-time* optimization in Haskell to remove intermediate list creations. Here the library performs pattern matching on the class of the transformer passed to the `add` method *at runtime* to decide if it can be fused with the previous one or not using hard-coded rules[4].

Stream output channels follow a similar design philosophy. For example, they support transformations to filter, map a function or buffer messages sent to the channel. They do not track the complexity because we have not yet encountered a use case where many output transformations are stacked together on an output channel.

The ability to apply higher-order transformations to both input and output channels, which is particularity of our library, benefits expressiveness and performance. For example, instead of inserting a process between a decoder and an encoder process, a process can just add decoding and encoding functions respectively to its input and output channels. These will apply the transformation functions directly to the messages carried in the underlying segments in a tight loop.

Message Poisoning Messages in transit may be dropped because they are filtered by a channel or because the input interface of a channel is poisoned while it is buffering messages that are pending delivery. Since messages may themselves reference channels interfaces, our poisoning scheme must know how to poison the channel interfaces carried inside these messages to ensure the proper propagation of the poisoning signal.

For convenience, the library uses Scala's *type classes* mechanism [Oliveira et al., 2010] to let the compiler pass implicitly a `Message` dictionary to functions that may need to poison messages:

```
abstract class Message[-A] {
  def poison(a:A, signal:Signal):Unit
}
```

The library takes care of providing sensible default values for standard type messages. This is for example the case for stream channel types themselves:

```
implicit def ichanIsMessage[A]:Message[IChan[A]] =
  new Message[IChan[A]] {
    def poison(ichan:IChan[A], signal:Signal):Unit = {
      ichan.poison(signal)
    }
  }

implicit def ochanIsMessage[A]:Message[OChan[A]] = ...
```

By default, user defined messages are "pure", i.e. the poison method does nothing and the messages will be simply garbage collected. This behavior may be overridden using Scala's standard

[4]Every fusion framework should come with a rigorous correctness proof, however carrying such a proof is not a trivial exercise, as explained by Coutts et al. and it would largely exceed the scope of this paper.

implicit prioritization mechanism, for example, by providing a specialized implicit definition in the companion object of a user defined message.

4.3 Process-Level Channel Interfaces

Process-level channel interfaces shown in Figure 9 are I/O interfaces manipulated from monadic threads (Section 3). They expose streaming, word-at-a-time and poisoning primitives. The description of these methods will be covered progressively in the various examples introduced later in this paper. End users invoke the `open` coordination primitives shown below to lift stream channels interfaces into process-level interfaces.

```
def open[A:Message](ichan:IChan[A]):IO[Input[A]] =
  new IO((t, k) => k(new Input(t, ichan)))

def open[A:Message](ochan:OChan[A]):IO[Output[A]] =
  new IO((t, k) => k(new Output(t, ochan)))
```

These interfaces keep track of the seed passed to continuation of stream channel interfaces on behalf of lightweight processes using a private variable. They also register themselves as resources to the user-level thread context passed as argument to their constructor such that they get automatically poisoned when the context is cleaned up. Alternatively, end users may invoke their `release()` method to get back the underlying stream channel and remove the interface from the resource control of the current process.

```
abstract class SelInput[+A] { //Selectable Input
  def read():IO[A]
  def foreach[B](f:A => IO[B]):IO[Unit]
  def <+>[B](right:SelInput[B]):SelInput[Either[A, B]]
}

class Input[+A:Message](
  t:UThreadContext, private[this] var ichan:IChan[A])
extends SelInput[A] extends Resource {
  def map[B:Message](f:A => B):Input[B] = ...
  def filter(p:A => Boolean):Input[A] = ...
  def span(p:A => Boolean):Input[A] = ...
  ...
  def release():IO[IChan[A]] = ...
  def poison(s:Signal = EOS):IO[Unit] = ...
}

class Output[-A:Message](
  t:UThreadContext, private[this] var ochan:OChan[A])
extends Resource {
  def write(a:A):IO[Unit] = ...
  def flush(in:Input[A]):IO[Unit] = ...
  def map[B:Message](f:B => A):Output[B] = ...
  ...
  def release():IO[OChan[A]] = ...
  def close(s:Signal = EOS):IO[Unit] = ...
}
```

Figure 9: Process-Level Channel Interfaces.

Buffered I/O Although process-level channel interfaces expose streaming primitives, sometimes it is convenient to process messages one at a time using imperative read and write primitives. To preserve the performance benefits of batching, process-level channels are buffered. Messages buffered on an output during write operations are flushed in a single segment once the output is closed or when the process is suspended. For instance, a process can be suspended during a read operation if it has consumed all the messages contained in the last segment buffered by that input.

4.4 An Example of Choice

We call the <+> operator defined on process-level input channels, the *input choice*, in analogy to the π-calculus + operator. The action:

```
(x <+> y).read() >>\ {
  case Left(v) => P(v)
  case Right(w) => Q(w)
}
```

will either read a value `v` from channel `x` and then behave like P or read a value `w` from channel `y` and behave like Q.

As observed by Peyton Jones et al. during the design of Concurrent Haskell [1996], the choice operator is expensive to implement, but we find it useful. For example, this operator can be used in conjunction with the one shot timer channel defined in Section 4.1 to define a method that reads a message within a given delay:

```
def readWithin[A](in:Input[A],
                  delay:Long, unit:TimeUnit
                 ):IO[Option[A]] = managed {
    open(TimerChan.after(delay, unit)) >>\ {timer =>
    (timer <+> in).read() map {
      _.fold(Function.const(None), Some(_))
    }}
}
```

This method attempts to read a message on an input, but returns none if a timeout occurs before a new message is available. Wrapping the action in a `managed` block is not mandatory. It only prevents that a timer task gets unnecessarily triggered if a message is available before the timeout by poisoning the timer channel.

5 Programming Model and Scheduling Strategies

In this section, we outline the process-oriented programming model we built upon the abstractions described in this paper. Secondly, we describe how we leveraged these abstractions to build a scheduler that eliminates unnecessary context switches when interactions in a process network occur sequentially.

In our model, component types are specified by defining the main method of an abstract *process type* class patterned after function types in Scala:

```
type Process[R] = (UThread, OChan[R]) => Unit

abstract class ProcessTypeixj[
      I1:Message, ..., Ii:Message,
      O1:Message, ..., Oj:Message, R:Message] {
  ...
  def apply(i1:IChan[I1], ..., oj:OChan[Oj]
  ):Process[R] = ...

  protected def main(i1:Input[I1], ..., oj:Output[Oj]
  ):IO[R]
}
```

By substituting the indices i and j for a number of inputs and outputs in the code template above we obtain the real code of a process type i.e. `ProcessType1x0`, `ProcessType0x1`, `ProcessType1x1`, etc. The abstract class is parameterized by the type I_i and O_j of the input and output channel interfaces passed as argument to the main method of a process, followed by its result type R. End users apply the factory method `apply` of a process type to stream interfaces to create a new process instance `Process` whose behavior is defined by the `main` method. The process instance will take care of opening the corresponding process-level interfaces on behalf of the end user before calling the `main` method. These interfaces will be automatically poisoned after the main action terminates thanks to the resource control mechanism described in Section 3.3.

A `Platform` takes care of creating a new user-level thread and a *result channel* for processes passed as argument to its `launch` method:

```
abstract class Platform {
  def launch[R:Message](p:Process[R]):IChan[R]
}
```

The result channel will carry either the final result or the uncaught exception signal that caused the termination of the main action. It is worth noting that monadic processes can launch new processes as well through the `platform` field referenced by their user-level thread:

```
def launch[R:Message](p:Process[R]):IO[IChan[R]] =
  new IO[IChan[R]]((t, k) => k(t.platform.launch(p)))
```

A platform can adopt different strategies for scheduling the tasks submitted to its user-level threads. The following describes the schedulers used in our evaluation.

Flow Parallel Scheduler (FP). The flow parallel scheduler exploits the structure of our design to maximize the utilization of native threads before they return to their underlying thread pool to execute another continuation task. The aim is to reduce the impact of factors that affect negatively the performance in multicore environments like contention on shared pool resources, context switches, and the likelihood to suspend and resume native threads when a new task is submitted. Our design makes use of two trampolines [Ganz et al., 1999] that let native threads execute sequential interactions without having to return to their thread pool. First, all the continuation tasks submitted to the user-level thread of a running process are executed sequentially using a first trampoline (T1) until it yields. A running process yields, or becomes suspended, once all its continuations are suspended. Secondly, the scheduling algorithm relies on the observation that during an input or output interaction on a channel, one process is running while the neighbour *may* be suspended. A reference to the continuation task submitted by a suspended neighbour process resumed during such interaction is temporarily held back in a thread-local variable (TLV) maintained by the native thread executing the running process. If the running process resumes another process before it yields, the task kept in the TLV is submitted to the thread pool for parallel execution and replaced by the continuation of the newly resumed process. Once the running process yields, the current native thread executes immediately a neighbour's continuation stored in its TLV without returning to its thread pool. This design ensures that if multiple processes are resumed, at least one process is resumed within the same thread. This way, the execution of an application remains purely sequential as long as one process is resumed at a time.

Naive Scheduler (NS). Each user-level thread created by this scheduler maintains also a local task queue to submit continuation tasks sequentially to their underlying executor. But, in contrast to the previous scheduler, both T1 and T2 trampolines are disabled and every continuation task is submitted to the underlying thread pool. This scheduler will be used in the benchmarks below to demonstrate the relative performance gains of the flow parallel scheduler.

6 Examples and Experimental Results

This section describes three novel implementations of classical examples used to showcase the expressiveness of concurrent programming frameworks: *a thread ring* [Halen et al., 1998], *a parallel genuine prime sieve* [Kahn and Macqueen, 1977] and *a chameneos-redux* [Kaiser and Pradat-Peyre, 2003]. We chose here popular examples that are well documented such that curious readers can easily lookup similar implementations in their favorite programming environment.

We benchmarked implementations based on Molecule, using either the flow parallel scheduler (Molecule/FP) or the naive scheduler (Molecule/NS), and implementations based on the standard Scala Actors library [Haller and Odersky, 2008] bundled with Scala 2.9.1. This last one is merely provided as reference because it shares the same programming language and execution platform as ours. A full comparison with the state-of-the-art on real world examples is left for future work as it would largely exceed the scope of this paper. All benchmarks were performed on a 2 times 12-cores 64-bit machine (AMD Opteron 6174, 2.2 GHz per core), running a Java HotSpot 1.7.0u4 Server VM configured with the NUMA-aware parallel garbage collector under Linux 3.3.0. Scala Actors and Molecule schedulers were configured to submit their continuations to the same implementation of the JSR166 fork/join pool [Lea, 2000]. We ran the same benchmarks using various pool sizes to measure how performance is affected by multicore execution. In each case we took the median of 5 runs.

6.1 The Thread Ring

The thread ring application consists in a network of P neighbour processes interconnected in a ring configuration by integer channels. Each process is configured with a distinct *label* indexing its position in the ring. A process forwards any integer value received on its input decremented by 1 to its downstream neighbour. The application starts by feeding a value N to the first process in the ring and ends by displaying the label of the process that receives 0 on its input. The code sample below implements a word-a-at-time version of a ring process, we call a node:

```
object Node extends
ProcessType2x2[Int, Int, Int, Int, Unit] {

  def main(label:Input[Int],
           in:Input[Int], out:Output[Int],
           result:Output[Int]):IO[Unit] = for {
    l <- label.read()
    _ <- in.foreach {i =>
           if (i > 0) out.write(i - 1)
           else result.write(l) >> out.close()
         }
  } yield ()
}
```

The node is parameterized by its label, its neighbour channels and a channel on which to forward its label once it finds 0. After the label has been written down, the process closes its output channel, which propagates the `EOS` signal to its neighbour process downstream. The `foreach` loop of the neighbour will then terminate, exactly as such loop behaves on regular lists. Since there is nothing else to do after the loop, the process terminates and poisons its channels, causing the next process to terminate, and so on, until the entire network collapses. The sample below shows a second implementation of the same component that uses streaming primitives:

```
object Node extends ProcessType2x2[...] {
  def main(...):IO[Unit] =
    out.flush(in.span(_ > 0).map(_ - 1)) >>
    unless (in.isEmpty) { result.flush(label) }
}

def unless(b:Boolean)(action: IO[Unit]):IO[Unit] =
  if (b) action else IO(())
```

The `span` primitive creates a temporary stream that ends with `EOS` once an elements does not satisfy its predicate. The `map` primitive is used here to decrement every value received on the temporary stream. When the predicate becomes false, the `span` primitive resets its input interface to the remaining of the stream. The `flush` streaming primitive forwards to its output every segment received on the input channel interface passed as argument. It returns control to the caller once the input stream has entirely been consumed. The process will then write its label only if the remaining stream contains one element, which can then only be 0. After, the process terminates and all its channels are poisoned automatically with the `EOS` signal. This propagates the termination downstream to its neighbour until the entire network collapses in a graceful manner. In both cases, the process networks terminate seamlessly as soon as one of the processes of the ring dies, i.e. without any explicit support from the application to propagate poison signals.

The thread ring is wired using the following test method:

```
def test(platform:Platform, P:Int, N:Int) {
  val first = Chan[Int]()
  val result = OChan.stdout

  val last = (1 until P).foldLeft(N :: first) {
    case (prev, label) =>
      val next   = Chan[Int]()
      platform.launch(Node(label, prev, next, result))
      next
  }
  platform.launch(Node(P, last, first, result))
  platform.collect()
}
```

The `Chan` object implements a factory method for cooperative channels. These inherit from both system-level input and output interfaces. The channel `SysOChan.stdout` encapsulates the JVM standard output into an system-level output. The standard output channel ignores poison signals; it remains perpetually open and can be shared safely between multiple processes. System-level channels are automatically lifted inside stream interfaces (c.f. Section 4), for instance when we call the :: operator to prepend a value in front of a channel. The `platform.collect()` method blocks the native bootstrap thread until all the processes launched on the platform have terminated.

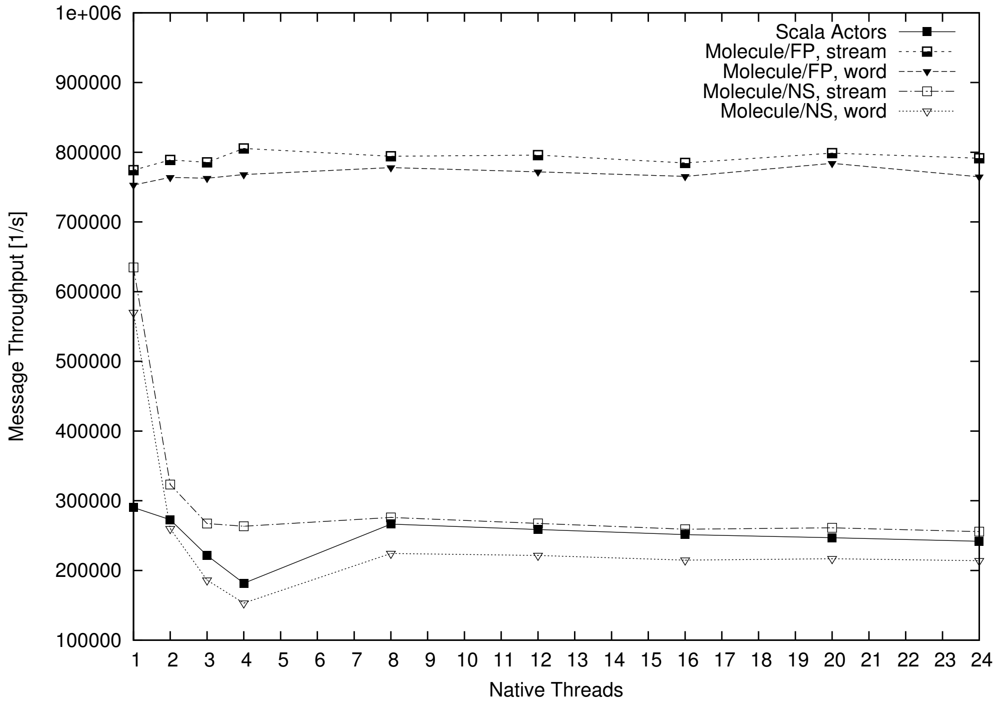

Figure 10: Thread ring benchmarks (P=503).

Figure 10 shows the throughput of message passing for a ring of 503 nodes while gradually increasing the number of native threads configured in the underlying platform. Benchmarks with Molecule/FP exhibit clearly the highest performance compared to Molecule/NS and Scala Actors. Since this process network never triggers more than one interaction at a time, Molecule/FP executes all the processes within the same thread using the trampolines described in Section 5. This is much faster because it eliminates entirely the negative impact of resuming and suspending threads sequentially. Indeed, in the other cases, above 1 thread, each I/O task submitted sequentially by the current thread is likely to wake up another thread in the underlying thread pool, which adds a significant overhead in this benchmark.

The difference between word-at-a-time and streaming versions is not particularly significant here because segments flowing through the ring never carry more than one message. Therefore, streaming primitives cannot take advantage of batching. Part of the overhead in the word-at-a-time version must be attributed to the extra level of indirection brought by the IO monad, which puts more pressure on the garbage collector. This is not the sole explanation because this constant overhead is slightly more important with Molecule/NS (50000 msgs/s in average), which submit every continuation to the fork/join pool, than with Molecule/FP (in 20000 msgs/s in average), which executes every continuation in its trampolines. This difference is interesting but we do not find it important enough to investigate further the causes in the scope of this paper.

The overhead of garbage collection is more apparent with Molecule/FP on small rings as seen on Figure 11, which compares the throughput of streaming and word-at-a-time versions while growing the ring from 0 to 80 nodes. It can be seen also that the throughput is much higher than for large rings and then drops sharply around 64 nodes to a similar level. Since our Opteron server has 64KB of Level-1 (L1) data cache and each node uses about 1KB of heap[5], or a bit less in the case of the streaming version, the drop occurs as soon as the heap space occupied by the ring does not fit in the L1 data cache, which happens around 64 nodes.

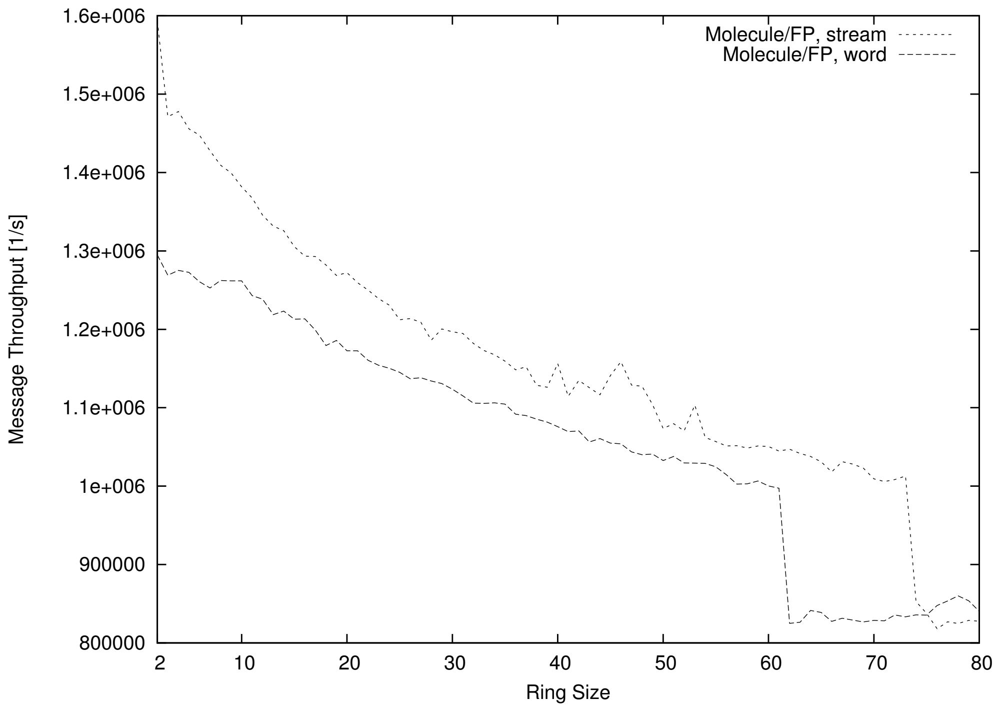

Figure 11: Cache effect in the thread ring.

6.2 The Parallel Genuine Prime Sieve

Many parallel implementations of this algorithm have been provided in the literature. Ours is special because of its support for termination and because prime numbers are filtered by channel transformations, and not by processes. Its implementation is the following:

[5]We confirmed this value by measuring the difference of heap space occupied by two rings of different size.

```
object Sieve
  extends ProcessType1x1[Int, Int, Unit] {
    def main(ints:Input[Int], log:Output[Int]) = for {
      prime <- ints.read() orCatch {
                 case EOS => shutdown(())
               }
      _     <- log.write(prime)
      _     <- handover(
                 Sieve(ints.filter(_ % prime != 0), log)
               )
    } yield ()
}
```

The process logs each newly discovered prime in a stream of increasing integers to its output channel until there is nothing else to read on the input stream. If there is a new prime, it recurses by launching a new instance of itself connected to the input channel, transformed to filter all the multiples of that prime, and the original output channel. This relies on an additional factory method on process types that takes the extra step of releasing stream channels from the current process first. The `handover` method is a coordination primitive like `launch`, excepted that the new process instance reuses the result channel of the current process, and the method does not return normally — it halts the execution and closes all the channels of the calling process. Therefore, there is never more than one process instance running in this application, only an increasing number of filter transformations stacked up on the input channel. If there is no more data to read on the input channel, the termination signal `EOS` is raised as a user-level exception and the `shutdown` method is invoked. The `shutdown` method is a type safe coordination primitive parameterized by the result of the process type:

```
def shutdown[R](r:R):IO[Nothing] = ...
```

It poisons all outputs and inputs with the `EOS` signal and then terminates the process immediately with the result passed as argument. The method testing the prime sieve application is simple:

```
def test(platform:Platform, N:Int) {
  val ints = IChan(2 to N)
  val log = OChan.stdout
  platform.launch(PrimeSieve(ints, log))
  platform.collect()
}
```

The stream of incrementing integers `ints` is obtained by applying a factory method present in the companion object of system-level input channels to a standard Scala range.

Figure 12 shows the evolution of the time it takes to find all primes between 2 and 150000 when we vary the Complexity Cutoff Threshold (CCT) and the Segment Size Threshold (SST) optimisation parameters configured within the runtime. Since this example stacks many stream transformations together and runs over a long stream, it can benefit from parallelization and batching. The CCT dictates how many synchronous filter transformations can be stacked on a channel before they are automatically parallelized by the library. The SST controls the size of the segments generated by the input channel of incrementing integers. Additionally, synchronous filter transformations applied to a channel are dynamically fused together by the library to eliminate the creation of intermediate segments. This relies on a built-in rule specifying that the fusion of two filter transformations with predicate p and q is equivalent to a single filter transformation whose predicate is the conjunction of p and q.

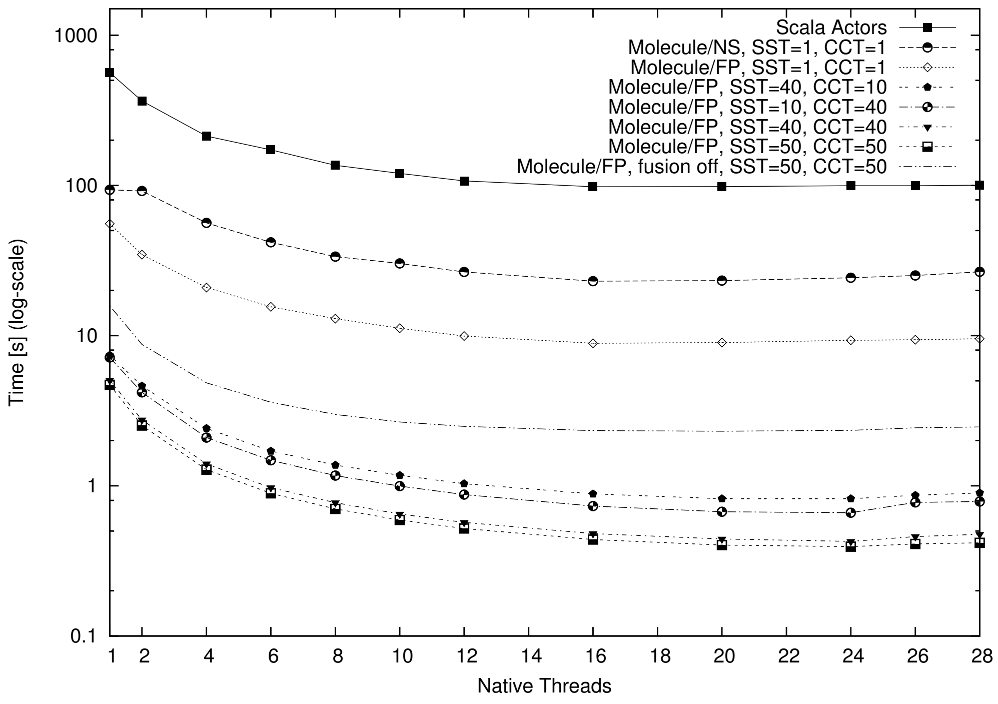

Figure 12: Genuine prime sieve benchmarks.

As expected, the implementation based on Scala Actors performs poorly in this benchmark because its runtime does not implement any of the optimizations presented in this paper. Even when all optimisations are turned off, that is when SST and CCT are equal to 1, Molecule/NS is about 4 times faster than Scala Actors. One reason is that streaming primitives, like the one used to filter integers, are extremely lightweight compared to actors and create little garbage. For the same settings (SST = 1, CCT = 1), we gain another factor 2.5 speedup by replacing Molecule/NS with Molecule/FP. Indeed, some messages will be entirely filtered out from the input stream and, depending on the asynchronism, some sections of the pipeline may become purely sequential for a while and thus the execution will benefit from the fast trampolines implemented by this scheduler.

Another means to reduce the context switching overhead is to increase CCT and SST values such that the application does more work per continuation task. Increasing these values permits to optimize significantly the execution of this example. We gain another 26 times speedup by setting them to 50, which is high enough to make context switching negligible. At its peak performance Molecule is 100 times faster than Scala Actors and 65 times faster than Molecule/NS unoptimized. Note that fusions played a significant role in reaching these high performance figures as shown in the benchmark where fusions are turned off.

6.3 The Chameneos-Redux

The chameneos-redux is a more sophisticated example of application that benefits from the expressiveness of strongly typed and first-class communication channel interfaces. This application is described by Kaiser and Pradat-Peyre [2003] along with its implementation in Java. We will not provide here the complete implementation but we will highlight the higher expressiveness resulting from the higher-order and type-safe communication primitives of Molecule and their embedding in Scala's type system.

Defining the protocol. Each Chameneo process sends a `MallRequest` message to the Mall process, which groups these requests by two. This request carries information about the Chameneo that initiated the request and a reply channel on which it waits for a response of type `ChameneoMessage`. A valid response from the Mall is either a `Meet` request or a `Copy` message. The first Chameneo in a group receives a `Meet` request whose reply channel is set to the reply channel of the second Chameneo in a group. It can then only reply with a `Copy` message, which tells the second Chameneo to copy its color. The code below shows how we have encoded this protocol in the Scala type system:

```
trait ReplyChannel[-A] {
  def replyCh:OChan[A]
}

case class ChameneoId(label:Int, color:Color)

case class MallRequest(id:ChameneoId)(
  val replyCh:OChan[ChameneoMessage]
) extends ReplyChannel[ChameneoMessage]

abstract class ChameneoMessage

case class Meet(peer:ChameneoId)(
  val replyCh:OChan[Copy]
) extends ChameneoMessage with ReplyChannel[Copy]

case class Copy(id:ChameneoId) extends ChameneoMessage
```

The messages carrying a reply channel are marked by the `ReplyChannel` trait. Note that although the types do not specify how many messages might be exchanged on a channel, they express clearly the ordering and the types of messages exchanged in a protocol.

Using streaming primitives to group chameneos. The Mall *lazily* groups by two a number of meeting requests received on the input interface `mallIn` of a *server channel* using the `grouped` and `take` streaming primitives, which are similar to those defined on regular lists in the Scala standard library:

```
def behave(n:Int, mallIn:Input[MallRequest]) =
  mallIn.grouped(2).take(n).foreach { mates =>
    val first :: second :: Nil = mates
    replyTo(first)(Meet(second.id)(second.replyCh))
  }
```

A server channel is a N-to-1 channel producing messages sent by multiple clients in the order in which they arrive. The Mall process, which can be assimilated to a signaling server, replies with the `id` and the reply channel of the second Chameneo request to the first one in the group. The `replyTo` method is a convenience method offered by the library to reply to requests, i.e. messages that carry a reply channel:

```
def replyTo[A:Message](req:ReplyChannel[A])(a:A) =
  open(req.replyCh) >>\ {out =>
  out.write(a) >> out.close()}
```

After it has arranged `n` meetings, the Mall process terminates and its input server channel interface `mallIn` is automatically poisoned. This poisons in turn all the requests the server channel may have buffered or might receive in the future. This includes their response channel thanks to the following default implicit definition offered by the library for request messages, which inherit from the `ReplyChannel` trait.

```
implicit def reqIsMsg[Rq <: ReplyChannel[_]] =
  new Message[Rq] {
    def poison(m:Rq, signal:Signal):Unit =
      m.replyCh.close(signal)
  }
```

Defining chameneos behavior. Chaneneo processes can send requests using the following convenience method offered by the library.

```
def requestTo[Rq <: ReplyChannel[Rp], Rp:Message]
    (out:Output[Rq])(mkReq:OChan[Rp] => Rq):IO[Rp] =
{
  val chan = Chan[Rp]()
  out.write(mkReq(chan)) >>
  open(chan:IChan[Rp]) >>\ {_.read()}
}
```

This sends a request to an output channel in the π-calculus style, i.e. by embedding a reply channel inside the request and then reading the response on the other end. A Chameneo can then send repeatedly requests to the the Mall using the output interface `mallOut` of the server channel and count the number of meetings in which it participated using the recursive loop defined below.

```
def behave(count:Int, id:ChameneoId):IO[Nothing] =
  {
    requestTo(mallOut)(MallRequest(id)) orCatch {
      case _ => shutdown(count)
    }
  } >>\ {
    case other@Meet(otherId) =>
      val newColor = complement(id.color, otherId.color)
      val newId    = id.copy(color = newColor)

      replyTo(other)(Copy(newId)) >>
      behave(count + 1, newId)

   case Copy(otherId) =>
      val newId = id.copy(color = otherId.color)

      behave(count +1, newId)
 }
```

As a call to `behave` is in tail call position within the monadic domain, it will not increase stack or heap usage (c.f. Section 3). A Chameneo will catch the poisoning of the reply channel of one of its requests as an exception once the Mall terminates. It will then shutdown itself by returning the number of chameneos it met as a result.

Using interleaved parallelism to collect the status of child processes The Chameneo results are collected by a supervisor process using the `parl` action, which collects the results of a list of actions into a list:

```
def parl[A](ios:List[IO[A]]):IO[List[A]] = ...

parl(chameneos).map{_.sum}
```

The `parl` action is implemented in a similar manner as the `par` combinator described in Section 3.2. Here, `chameneos` is a list of actions where each action launches a Chameneo process and then reads the number of meetings in which it participated on its result channel.

Benchmarking Figure 13 compares the performance of Scala Actors and Molecule/NS for 300 chameneos and 300000 meetings as we increase the number of native threads in the underlying platform. The SST threshold controls here the maximum number of requests that can be read in a single segment by the Mall process on its server channel.

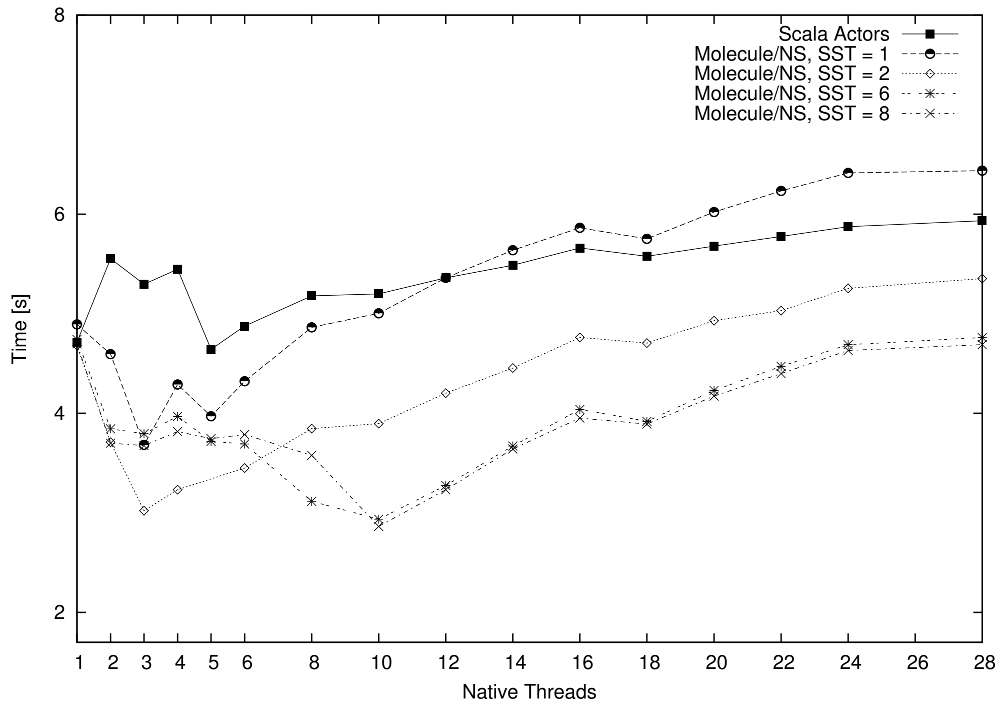

Figure 13: Chameneos-Redux benchmarks: Scala Actors vs. Molecule/NS.

This application does not scale well in multicore environment because the Mall process is shared amongst all chameneos and hence the main bottleneck. The contention created by sending concurrent requests to the Mall quickly starts to overcome the benefits of parallelization above a few threads until there are as many threads as cores available in the underlying hardware (24). The same pattern can be observed in each of these benchmarks. When the SST is set to 1, the contention dominates above 3 threads. We attribute the fact that the contention increases faster with Molecule/NS than with Scala Actors to the implementation of our server channel. We can speedup the average execution time by increasing the SST threshold to reduce the amount of context switches. We obtain the best timings in average when the SST is set to 6, which lets the Mall dispatch up to 3 parallel meetings in a single task. Increasing the SST further to 8 does not bring significant improvements anymore in this example. The performance is then optimal with 10 threads where it is almost 2 times faster than the unoptimized version (SST = 1). However, one could argue whether it is worth spending 7 additional threads for a negligible improvement compared to when SST is set to 2. With this last setting, a performance close to the optimum is attained with only 3 threads.

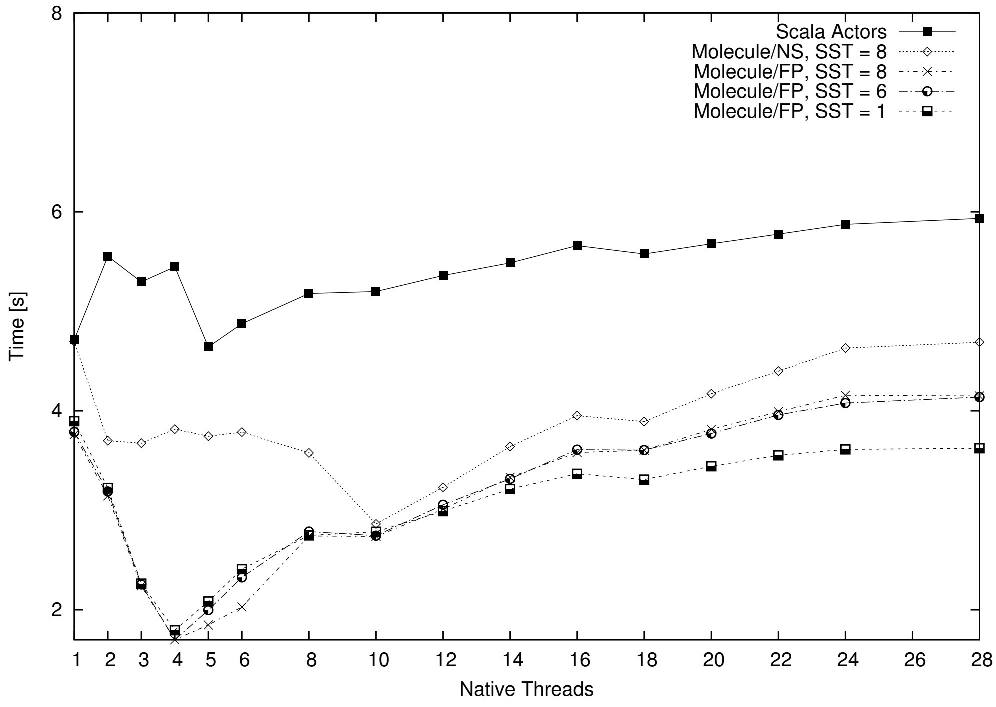

Figure 14: Chameneos-Redux benchmaks: Scala Actors, Molecule/NS vs. Molecule/FP.

As seen in Figure 14, the flow parallel scheduler yields better results by reducing the impact of context switches. In fact, we measured that it captures the submission of two tasks on three in its trampolines for this application. This occurs essentially during the sequential color exchange between pairs of chameneos. In each case, the optimal performance is attained with 4 threads, independently from the SST value. There, Molecule/FP is 2 times faster than Molecule/NS and 3 times faster than Scala Actors. Given the FP scheduler executes so many tasks sequentially already, increasing the SST threshold has little effect. Slightly annoyingly, although the impact is limited, increasing this threshold affects negatively the performance above 10 threads. Providing a deep analysis of the dynamics of this complex feedback system is beyond the scope of this document. However, we guess that setting the SST low throttles down the number of meetings triggered at the same time which reduces contention on the other side of the server queue. Thankfully, increasing the SST value beyond 6 does not harm performance further (16% in the worst case) and, even though it cannot alleviate the contention issue inherent to this kind of application, the flow parallel scheduler yields systematically the most interesting results.

7 Related Work

The topics of asynchronous and concurrent programming have given rise to a vast literature. Some work focuses exclusively on language-level extensions to ease asynchronous programming with callbacks and leaves the definition of a high-level concurrent programming model open for library developers. In absence of coroutines, many solutions apply some variation of a CPS transformation to the source code in a manner that blends well with the host languages features, such as exception handling, resource control (only on .NET) and built-in control structures. F# asynchronous workflows [Syme et al., 2011], C# async [Torgersen, 2010] and Scala's continuation plugin [Rompf et al., 2009] fall in this category. The F# approach based on computation expressions bears many similarities with how Molecule leverages Scala's for-comprehension to desugarize monadic expressions. C# async, on the other hand, depends on the generation of state machines, which is harder to maintain. On the JVM, Scala's continuation plugin implements a selective CPS transform, which is driven entirely by effect-annotated types. The solution uses first-class polymorphic delimited continuations, which can be composed in a flexible manner at application-level using 'shift' and 'reset' operations devised by Danvy and Filinski [1992]. A definite advantage over Molecule's application-level monad is its support for effect tracking. However, we find that exposing the type of our monad in method signatures is more natural and offers more valuable information to application developers to distinguish and compose asynchronous control flows compared to manipulating annotated types using Danvy and Filinski operators. Performance wise, the major bottleneck observed in our benchmarks in absence of contention is due to context switching, which is inherently higher during word-at-a-time interactions, and less by the extra level of indirection introduced by monadic I/O, which creates small and short lived immutable objects that can be efficiently garbage collected.

Li and Zdancewic [2007] designed an application-level continuation monad in Concurrent Haskell [Peyton Jones et al., 1996] to schedule concurrent interactions between monadic threads and system-level interfaces, in a way that scales to real-world network services. Their monadic threads support exceptions and asynchronous I/O but not streaming I/O. Although streaming I/O was found awkward in Concurrent Haskell and dropped in favor of monadic I/O, they cohabit well together in our programming model.

Many libraries offer a high-level concurrent programming model based on the Actor model on the JVM [Karmani et al., 2009] but none of them feature higher-order streaming primitives or message poisoning. Kilim [Srinivasan, 2010] and Scala Actors [Haller and Odersky, 2008] are two modern and representative approaches that come the closest to Molecule in terms of features. Both of them work on unmodified JVMs and offer lightweight communication primitives built upon continuations. Kilim is a concurrent programming library for Java. Unlike Molecule, which uses pure library approach, Kilim depends on a CPS bytecode post-processor for its user-level threads, called Fibers. Its Actors communicate using one or more type-safe message mailboxes. These are mobile and may suspend senders by holding back their continuation. However, Kilim does not offer any support for termination. Like Molecule, Scala Actors is a pure DSEL library written in Scala. Scala Actors and Erlang [Virding et al., 1996] share a similar programming model. In contrast to Molecule, each Actor features a single mailbox and communicate using asynchronous messaging and untyped communication primitives. An Actor may catch termination notifications from other Actors. However, the termination links must be wired explicitly. Moreover, as opposed to Molecule and Erlang, which offer a uniform lightweight threaded programming model, Scala Actors uses a hybrid model that solves half the problem with callbacks. While it can support an `andThen` combinator to sequence in a modular way non-blocking interactions using a mutable variable and control exceptions, the same mechanism cannot thread the result of a non-blocking interaction to the next one in a modular manner like the monadic `bind` operator.

JCSP [Sputh and Allen, 2005] and CHP [Brown, 2008] are two examples of process-oriented libraries written respectively in plain Java and Haskell. Both draw on Hoare's CSP model and, as such, do not feature mobile channels. CHP's standard library offers reusable higher-order processes, which implement classical 'map' and 'filter' functions on top of read/write primitives, but no higher-order channel interfaces. Both libraries offer a graceful termination mechanism to terminate process networks by spreading a poison signals. However, they don't implement resource control facilities. Therefore, they rely on the application developer to catch and propagate the poison explicitely within exception clauses.

Occam-π [Welch and Barnes, 2005] is an efficient and safe binding of key elements from Hoare's CSP and Milner's π-calculus into a programming language of industrial strength. Occam-π implements synchronous messaging and distinguishes I/O modalities on its mobile channel interfaces. It compiles down to native code and implements lightweight threads. In contrast to Molecule, it does not feature resource control, message poisoning or higher-order channel interfaces. Its CCSP scheduler [Ritson et al., 2009] exploits dynamically application-level communication patterns like Molecule's flow parallel scheduler. The design of the flow parallel scheduler is more modular and simpler: it is orthogonal to batching and delegates parallel task execution to an external thread pool, e.g. the JSR166 fork-join pool used in our benchmarks, which implements a work-stealing algorithm.

There is a vast body of related work on process-oriented programming in other languages and platforms [Sampson, 2010] as well. Many of these support the dynamic creation of processes and their runtime reconnection but, to the best of our knowledge, Molecule is the only one to offer support for seamless termination and higher-order streaming primitives on both its input and output channel interfaces.

8 Conclusion

We presented Molecule, a practical model for composing reconfigurable process networks that grow or shrink dynamically and its high-performance application to multicore programming hosted as a DSEL library in the Scala programming language on the JVM. The hybrid support for functional and imperative programming in Scala let us address both high-level expressiveness and low-level runtime issues in our programming model.

Our DSEL leverages two language abstractions originally developed separately for lazy and pure functional programming languages, monadic and streaming I/O, but combines them in a novel manner into a rich process-oriented programming model on the JVM. The clarity in the novel implementation of the classical examples stems from the use of higher-order streaming primitives, resource control integrated with user-level exception handlers, and the use of type-safe channels that guide the design of sophisticated protocols involving channel mobility. Since we retain the ability to use word-at-a-time read or write primitives on channels, in addition to higher-order streaming primitives, end users have the freedom to select and combine both imperative and functional styles in the manner they find the most appropriate.

Our runtime exploits the dynamics of application-level abstractions that capture the execution of user-level threads and their interaction over communication channels to mitigate transparently the cost of context switching faced by process-oriented designs in multicore settings. Next to runtime optimizations like message batching, fusions and parallelization of consecutive stream transformations, the flow parallel scheduler contributed systematically to improve the performance of our examples by restricting dynamically the parallelism engendered by continuation tasks whenever the communication is purely sequential. Experimental results show up to a 65 times speedup depending on the dynamics of the hosted application, as illustrated throughout the classical examples studied in this paper. Since real-world concurrent applications are likely to combine algorithmic and dynamic aspects of these examples, chances are that they may reap similar expressiveness and performance benefits.

Acknowlegdements

We would like to thank the (anonymous) reviewers for a considerable number of constructive comments, useful questions and suggested references.

References

Agha, G. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

Anderson, T., Bershad, B., Lazowska, E., and Levy, H. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10* (Feb. 1992), 53–79.

Backus, J. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

Banker, R., Davis, G., and Slaughter, S. Software development practices, software complexity, and software maintenance performance: a field study. *Manage. Sci. 44* (Apr. 1998), 433–450.

Brown, N. C. C. Communicating Haskell Processes: Composable Explicit Concurrency using Monads. In *Communicating Processes Architecture 2008* (Sept. 2008), pp. 67–83.

Clinger, W., Hartheimer, A., and Ost, E. Implementation strategies for continuations. In *Proceedings of the 1988 ACM conference on LISP and functional programming* (July 1988), pp. 124–131.

Coutts, D., Leshchinskiy, R., and Stewart, D. Stream fusion: from lists to streams to nothing at all. *SIGPLAN Not. 42* (Oct. 2007), 315–326.

Danvy, O., and Filinski, A. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science 2*, 4 (1992), 361–391.

Ganz, S., Friedman, D., and Wand, M. Trampolined style. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming* (1999), pp. 18–27.

Gelernter, D., and Carriero, N. Coordination languages and their significance. *Commun. ACM 35* (Feb. 1992), 97–107.

Halen, J., Karlsson, R., and Nilsson, M. Performance measurements of threads in Java and processes in Erlang. Available at http://www.sics.se/~joe/ericsson/du98024.html, Last visit April 2012.

Haller, P., and Odersky, M. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* (2008).

Haller, P., and Odersky, M. Capabilities for Uniqueness and Borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming* (2010).

Hewitt, C., Bishop, P., and Steiger, R. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence* (1973), pp. 235–245.

Hilderink, G. *Managing Complexity of Control Software through Concurrency.* PhD thesis, University of Twente, May 2005.

Hoare, C. A. R. Communicating sequential processes. *Commun. ACM 21* (Aug. 1978), 666–677.

Hudak, P. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse* (1998), pp. 134–.

Jones, M., and Hudak, P. Implicit and Explicit Parallel Programming in Haskell. Tech. Rep. YALEU/DCS/RR-982, Department of Computer Science, Yale University, Aug 1993.

Kahn, G. The Semantics of a Simple Language for Parallel Programming. In *Information Processing '74: Proceedings of the IFIP Congress.* North-Holland, Aug. 1974, pp. 471–475.

KAHN, G., AND MACQUEEN, D. Coroutines and Networks of Parallel Processes. In *Information Processing '77: Proceedings of the IFIP Congress*. North-Holland, 1977, pp. 993–998.

KAISER, C., AND PRADAT-PEYRE, J. Chameneos, a concurrency game for Java, Ada and others. *Int. Conf. ACS/IEEE AICCSA* (2003).

KARMANI, R. K., SHALI, A., AND AGHA, G. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (2009), pp. 11–20.

LEA, D. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (2000), pp. 36–43.

LI, P., AND ZDANCEWIC, S. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (2007), pp. 189–199.

MANSON, J., PUGH, W., AND ADVE, S. V. The Java memory model. *SIGPLAN Not. 40* (Jan. 2005), 378–391.

MILNER, R. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

ODERSKY, M., SPOON, L., AND VENNERS, B. *Programming in Scala, Second Edition*. Artima, 2011.

OLIVEIRA, B. C., MOORS, A., AND ODERSKY, M. Type classes as objects and implicits. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (2010), pp. 341–360.

PEYTON JONES, S., GORDON, A., AND FINNE, S. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages* (1996), pp. 295–308.

PEYTON JONES, S., AND HUGHES, J. Report on the programming language Haskell 98, a non-strict, purely functional language. Tech. rep., Haskell comittee, 1999.

PEYTON JONES, S., AND WADLER, P. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1993), pp. 71–84.

RITSON, C., SAMPSON, A., AND BARNES, F. Multicore scheduling for lightweight communicating processes. In *Proceedings of the 11th international conference on Coordination Models and Languages* (2009), pp. 163–183.

ROMPF, T., MAIER, I., AND ODERSKY, M. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming* (2009).

RYTZ, L., ODERSKY, M., AND HALLER, P. Lightweight Polymorphic Effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (2012).

SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2* (Nov. 1984), 277–288.

SAMPSON, A. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, Oct. 2010.

SPUTH, B. H. C., AND ALLEN, A. R. JCSP-Poison: Safe termination of CSP process networks. In *Proceedings of the 28th conference on Communicating Process Architectures* (2005), pp. 71–107.

SRINIVASAN, S. Kilim: A server framework with lightweight actors, isolation types and zero-copy messaging. Tech. Rep. UCAM-CL-TR-769, University of Cambridge, Computer Laboratory, Feb. 2010.

SYME, D., PETRICEK, T., AND LOMOV, D. The F# Asynchronous Programming Model. In *Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages* (2011), pp. 175–189.

TORGERSEN, M. Asynchronous Programming in C# and Visual Basic. White paper, Microsoft, Oct. 2010. Available online.

VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., 1996.

VON BEHREN, J. R., CONDIT, J., AND BREWER, E. A. Why Events Are a Bad Idea (for High-Concurrency Servers). In *Proceedings of the 2003 HotOS Workshop* (May 2003), pp. 19–24.

WADLER, P. Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci. 73* (Jan.

1988), 231–248.

Wadler, P. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (1995), pp. 24–52.

Wadler, P. An angry half-dozen. *SIGPLAN Not. 33* (Feb. 1998), 25–30.

Welch, P., and Barnes, F. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP* (Apr. 2005), pp. 175–210.

Welch, P. H. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10* (Apr. 1989), pp. 310–317.