Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

A scalable runtime system for type-driven concurrent programming

Author:
Elias Benussi

Supervisor:
Nobuko Yoshida

Submitted in partial fulfilment of the requirements for the Computing MEng degree of Imperial College London

June 2018

Abstract

Concurrent programming using shared memory and locking is known to be extremely error prone, which is why more and more frameworks and tool-kits are providing interfaces encouraging concurrency through communication. Effpi is an experimental, theoretically-grounded toolkit for type-driven concurrent programming. It provides expressive *types* that specify the behaviour of concurrent agents, and *domain-specific languages (DSLs)* for writing message-passing programs, in a style reminiscent of well-established toolkits like Erlang/OTP and Akka. The Effpi types and DSLs are embedded in the Scala programming language: hence, the Scala type checker can statically verify whether an Effpi program conforms to an Effpi type, and this ensures that a program runs and interacts as specified by its type.

However, the original Effpi implementation is based on a simple runtime system that executes each concurrent agent in a dedicated operating system thread. This approach is inefficient and does not scale beyond a few thousand agents/threads. For this very reason, toolkits like Erlang/OTP and Akka have sophisticated runtime systems that use a small number of OS threads, an yet, can execute applications with *hundreds of thousands* of concurrent agents.

This project presents a new runtime system for Effpi, that decouples concurrent agents from OS threads. Our new runtime greatly improves the performance of Effpi programs: our benchmarks show an improvement in scalability of over an order of magnitude, and an increased speed of execution of up to a factor of 100. Our new runtime can successfully handle applications with hundreds of thousands of agents; moreover, our new runtime does not alter the types and DSLs originally provided by Effpi, and thus preserves its theoretically-grounded properties.

Acknowledgments

I would like to thank my supervisor, Professor Nobuko Yoshida, for the opportunity she has given me with this project and for her constant encouragement to strive for excellence. Special thanks also go to her research group member, Dr Alceste Scalas, for the support he offered me over countless hours of meetings, both in person and via Skype. His insight and mentoring have been invaluable and I am extremely grateful for all the time he has dedicated to me.

To my friends Domenico, Abraão, Michael and Giulio, thank you for making my life hell when I am happy and bearable when I am sad. Also to my best friend, Youssef, for his positive attitude and constant encouragement throughout these past years.

I am especially grateful to my parents, Andreina and Fulvio, and the rest of my family for the incessant support over the years. Most of what I do would not be possible nor meaningful without you.

Contents

Chapter 1

Introduction

Do not communicate by sharing memory; instead, share memory by communicating [9]

This is a maxim I got used to hearing throughout my years at university. It refers to a general guideline in concurrent programming, advising to create concurrent units with their own local memory which communicate with one another to share and distribute information, as opposed to having a central, shared storage which is accessed by many concurrent threads. The explicit communication makes it easier to trace the information flow, which in turn helps with avoiding deadlocks, which are instead easy to encounter in concurrent programs based on shared memory.

Two notable theories that study concurrency through communication are the actor model and π-calculus. Both resort to the definition of units of concurrent execution (actors and processes, respectively) that manage to share information by sending and receiving messages to each other. By design these strategies encourage the development of concurrent programs that avoid the common pitfalls of shared memory concurrency and manual locking.

For the actor model, there are famous and well established implementations such as Erlang [25] and Akka [16], but also newer, cutting-edge implementations such as Pony [26]. Erlang, for example, has been successfully used to create large scale concurrent systems by companies such as WhatsApp for its messaging servers [4] [11], Pivotal for its open source message broker RabbitMQ[22] and Amazon for SimpleDB (part of its extremely popular service, EC2)[10].

Implementations based on π-calculus on the other hand are less common. One such example is Effpi[24], an experimental framework for concurrent programming whose API provides two Domain-specific languages (DSLs): one that reflects the common syntax of π-calculus and an auxiliary one that reflects the actor model.

These DSLs are deeply embedded[27] in the Scala [1] programming language, and their types allow us to describe in details the behaviour of concurrent programs. Deep embedding ensures that the type checker can be leveraged to ensure that a program runs according to the behaviour described by the types.

1.1 Motivation

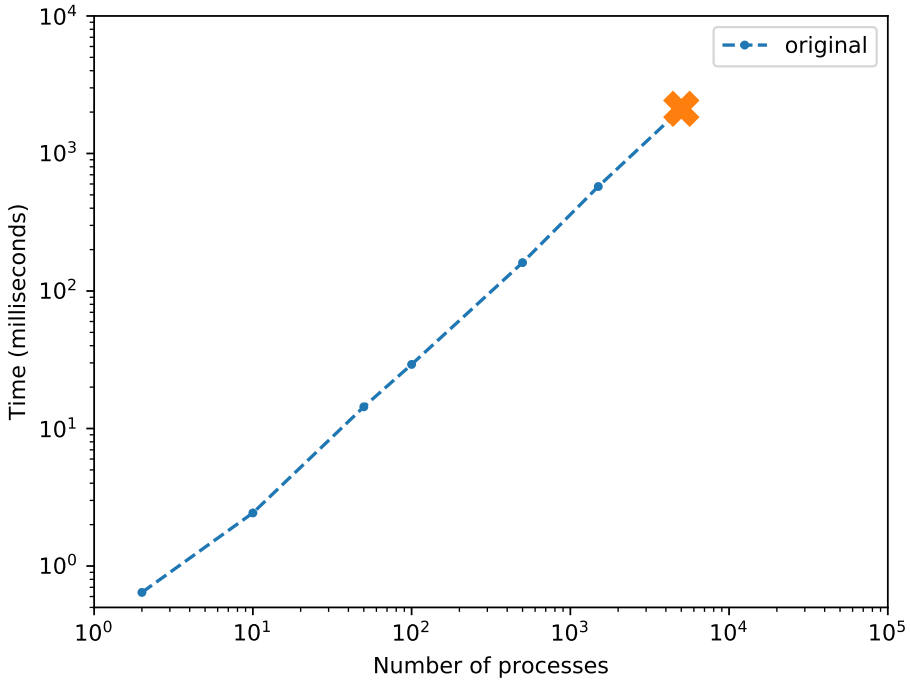


Figure 1.1: Example of the original runtime crashing, from the ForkJoin Throughput benchmark, described in details in Section 5.3.

Unfortunately the original implementation of Effpi has a naive runtime that creates a JVM thread per actor/process[2], which greatly undermines the ability to spawn them in large number, as illustrated in Figure 1.1. While it is still possible to benefit from the correctness checks even in the current state, for small concurrent systems with relatively few agents, it would be extremely helpful to be able to use Effpi to create large concurrent systems.

All the implementations of the actor model mentioned above actively decouple the concept of actor from that of operating system thread, making them extremely lightweight and far easier to manage. Consequently, all these implementations allow

[1]This project is actually implemented using the experimental Dotty compiler [21], which was necessary to implement the *dependent function types* described in Section 2.5

[2]From now on, unless otherwise stated, process will refer to an effpi-process as opposed to an operating system process

for the creation systems that can realistically handle hundreds of thousands of actors, something that would be impossible if they were internally represented as OS threads. This makes them very suitable for representing highly concurrent systems, and is a major factor behind their successful adoption.

Improving Effpi's runtime would help reduce the gap in performance with frameworks such as Akka, Akka Typed and Erlang, hence allowing a more engaging comparison using similarly large benchmarks which would enable to truly appreciate the extra safety features that Effpi provides (described in [24]). An improved runtime would also be paramount to encourage the adoption of Effpi in real world application where performance would likely be a criterion on which to choose a framework and would therefore influence its spread.

1.2 Objectives

The primary objective of this project is to improve the performance of Effpi's runtime. However changes made to the runtime should not compromise the correctness guarantees provided by the original implementation. Furthermore, the behaviour observed for a given system of processes should be independent of the implementation of the runtime, i.e. ignoring any difference due to interleaving.

In practice this entails the following goals:

1. **Preserving correctness:** Effpi's DSL and its types provide static guarantees on the correctness of the programs they are used to implement. Therefore our goal is to minimise the changes required to it while providing Effpi with a new runtime design, in order to be able to provide the same correctness guarantees.

2. **Scalability:** The original runtime has a naive implementation that leads to a very limited upper bound on the number of processes Effpi can spawn before the program becomes non-responsive. We need to implement a new runtime with an increased limit.

3. **Performance:** Improve the speed of execution and throughput of the runtime compared to the original implementation. We expect this to be (in part) a by-product of the reduction of JVM threads.

4. **Customisability:** Provide customisability of default parameters of the framework to allow the users to fine tune the framework to specific use cases. For example make it possible to specify how many JVM threads to spawn per physical core on the host machine, in order to optimise performance.

1.3 Challenges

- The optimised implementations of the runtime rely on the creation of an OS-like scheduling system for the execution of processes, that decouples effpi-processes from JVM threads. Since this implies multi-threaded access of a common scheduling queue (or an equivalent data structure) this will have non-trivial correctness implications and high risk of deadlocks (initially adopted designs, later abandoned for this reason can be found in Section 4.3.9 and Section 4.6).
- Since another objective is to improve speed of execution, further attention had to be paid to improve the efficiency in handling blocking input processes.
- The optimised implementations of the runtime introduced were tested on a number of benchmarks, in order to compare them to the original runtime implementation, and to asses their relative strengths and weaknesses. This process was also challenging, because it order to test the original runtime, we had to run it with systems too large for it to handle and pushed to its limit where programs would become non-responsive, to the point of causing the JVM to crash. Isolating the crashed state and recovering useful quantitative information in this scenario turned out to be problematic.

1.4 Contributions

The main contributions of this project are:

- In Section 4.2 we introduce a new runtime for Effpi. It uses a limited number of JVM threads and an OS-like scheduling system to govern when to run and, unlike the original runtime, when to suspend effpi-processes. The suspended processes, and all the information about their state required for running, are kept in a queue in memory until ready to be executed again by a JVM thread.
- In Sections 4.3 and 4.4 we present an optimised implementation of our runtime, introducing a separate scheduling system for input processes, which allows it to reduce the time spent scheduling processes that are not ready to be executed–e.g. blocked reading from a channel that has no data available.
- In Section 4.5 we describe another optimised implementation of the runtime that reduces the time spent by the runtime context switching between processes.
- In Section 5.3 we provide a series of benchmarks that validate the improvements of our optimised implementations compared to the original runtime (a short extract shown in Figure 1.2). We also compare the relative strengths

and weaknesses of our implementations and analyse the causes of these differences.

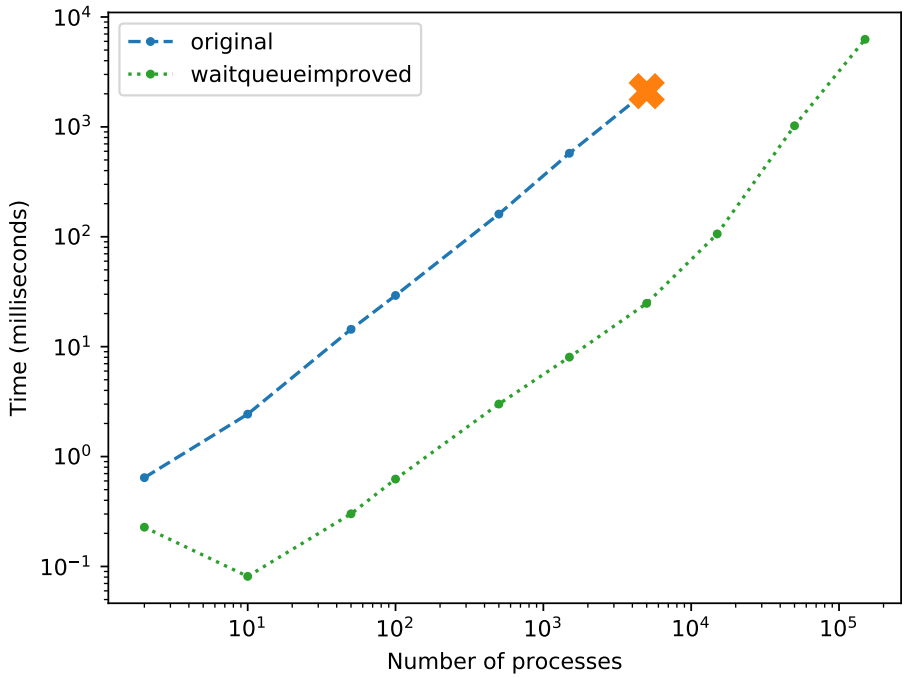


Figure 1.2: Example of successful results, from the ForkJoin Throughput benchmark, described in details in Section 5.3.

Chapter 2

Background: π-calculus and types

In this chapter we provide a brief overview of languages and types for concurrency, providing the theoretical background of Effpi (overviewed in Section 3).

In Section 2.2 and Section 2.3, we summarise the π-calculus and session types respectively: this sets the scene on how concurrent programs can be formalised, and their behaviour described in details and checked using type judgements. Then, in Section 2.4 we introduce the higher-order -calculus, and in Section 2.5 we describe the behavioural types that allow the description of full processes: this is the direct theoretical foundation of Effpi.

2.1 Introducing type-driven concurrent programming

As previously mentioned, implementing distributed concurrent systems is extremely error prone and difficult to test and verify. Often a developer will neglect important aspects of the internal interactions within the system in an effort to focus on specific implementation details. These errors unfortunately can be extremely hard to identify and fix, and they range from data races and deadlocks, to significant yet hidden errors in the implementation of the intended protocol of communication. Currently, many of the tools used to guide developers during the implementation of concurrent programs trying to verify their correctness are hardly satisfying. Some are too centralised and struggle with larger applications, while some have restricted specification methods, and many rely on run time checks that cannot give guarantees of correctness [5]. The methodologies used in this project revolve around *π-calculus* and *session types*. π-calculus is a technique used to formalise distributed concurrent systems and the communication between their agents. Session types in turn help verify the systems' run time correctness through local semantic checks at compile time [6]. In the coming sections we provide an intuitive overview of how these techniques can be helpful starting with the fundamentals of π-calculus.

2.2 π-calculus

π-calculus is dedicated to the description of communication protocols between agents in a system, similarly to how Turing machines and λ-calculus describe sequential computation [3]. The interactions between processes are formalised in terms of message passing (as opposed to synchronisation). This is done by defining primitives that are building blocks for communication between processes. The two fundamental entities in this calculus are values and channels. The main difference between the two is the ability of channels to act as medium of communication between processes. Values (and/or channels themselves) can be sent or received between processes through channels. While there are a number of slightly different π-calculi we will settle for the simplest syntax that is sufficiently powerful to give enough insight into the work done in the project (Table 2.1).

$P, Q ::=$	
0	*Nil process*
$P \mid Q$	*Parallel composition of two processes. They run asynchronously*
$u(x).P$	*Input of x on channel u with continuation P (x scoped in P)*
$\overline{u}\langle v\rangle.P$	*Output of v on channel u with continuation P*
$(\nu\ v)P$	*Generation of name v with scope in P*
$!P$	*Infinite parallel composition of P: $P \mid P \mid P \mid ...$*
$k \triangleright \{l_i : P_i\}$	*Branching gives options to choose from*
$k \triangleleft l.P$	*Selection of a particular label and continuation*
`def` D `in` P	*Recursive definition*
$X\langle e\rangle$	*Recursive call*
`if` e `then` P `else` Q	*Conditional*

Table 2.1: Syntax of the π-calculus used in this Section

2.2.1 Example I: Sending and receiving

In order to introduce π-calculus and its syntax we will start with a simple interaction between a user and a remote tool that performs numerical computations given an input and then returns the result. Given that technically our syntax does not provide arithmetic operations, and that therefore this is somewhat abusing it we will imagine a simple tool that calculates the successor of a number to keep it simple. Therefore if the user sends 42 to the remote tool she will get back 43. Using the syntax above this interaction can be expressed as:

$$\overline{a}\langle 42\rangle.a(y).0 \mid a(x).\overline{x}\langle x+1\rangle.0$$

Here user (left) and tool (right) are represented as two parallel sub-processes in the system. The former uses a public channel a to communicate the number to the latter, which then in turn sends back the result of the computation.

2.2.2 Example II: Private communication

Lets now consider a simple interaction in which Alice wants to tell Bob a secret. Assuming they both find themselves in a public setting, we assume that somehow they booth need to know that a secret is about to be told and that they agree on a secure way of communicating this secret, some *private channel*; once this is done the secret can be communicated safely. If these aspects can be formalised then the key features of the communication are encapsulated and the formalisation will be a good abstraction. Using our π-calculus, this can be expressed as:

$$(\nu\ u)\overline{a}\langle u\rangle.u(y).\overline{y}\langle c\rangle.0 \mid a(x).(\nu\ v)\overline{x}\langle v\rangle.v(y).0$$

Firstly we can notice that there are two subprocesses running in parallel, this will represent Alice and Bob. Alice communicates to Bob over the public channel *a* the channel *u* that it just created. Over this channel *u* Bob accepts to receive the secret and provides a private channel *v* of his own. Since this calculus relies on message passing, and this means sharing by communicating, the channels *v* and *u* will be private to Alice and Bob and invisible to anyone else. At this point the secret is communicated and then the interaction terminates successfully. This initial exchange of private channels can be thought essentially as a handshake. Since it is up to Alice to decide whether to tell the secret or not, she will have to initiate the communication. Only then can Bob provide back a private channel for the communication.

These covers the most basic elements of the syntax. We will now show that by using some of the remaining syntactic elements provided we can model increasingly complex systems.

2.2.3 Example III: Forwarder

The formula that we will introduce now describes a forwarder, a process that can be used to forward a new value to a given channel. This is part of a number of stand alone processes called small agents. These can be thought as design patterns in programming, as they are efficient constructs that achieve a common task.

$$NN(a) \stackrel{\text{def}}{=} !a(x).(\nu\ b).\overline{x}\langle b\rangle.0$$

As introduced above, *NN* can repeatedly produce a subprocess that accepts a private channel of communication *x* over a public channel *a* and then sends a newly created name *b* back through *x*.

It is possible now to see how π-calculus can help abstracting communication systems. However, while it is very powerful in what it allows us to model, it does not prevent us from modelling erroneous protocols by construction. Just as before introducing π-calculus we could come up with systems with incorrect message passing, deadlock and data races, so we can now, even using these formalisations. This becomes a problem when modelling more realistic and complex protocols, such as the Three Buyers Protocol that we are about to introduce.

2.2.4 Example IV: Three Buyers Protocol

The protocols illustrated by the execution in Figure 2.1 can be found with a detailed explanation in [6]. In short, it handles a transaction in which Alice wants to buy something from s Seller. If necessary, she can ask Bob for financial help, who in turn can ask Carol. Based on the request they will decide if they can contribute. If in the end they collectively have enough money to make the purchase, the seller is notified.

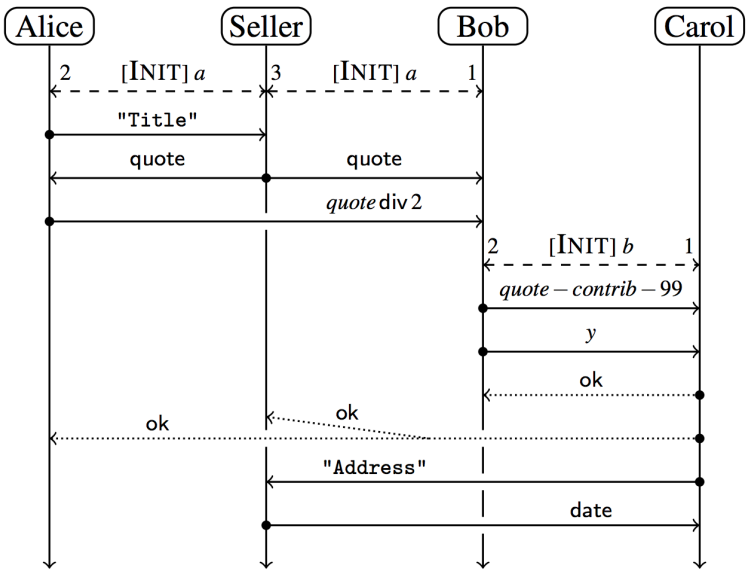


Figure 2.1: An execution of the three buyers protocol from [6]

Compared to industry level systems, this is still extremely simple. However it is complex enough to be able to picture how easily one could make a mistake in the

π-calculus description. This means our tool for formalisation is not powerful enough to prevent a variety of errors, such as type error of the message (string instead of number), deadlock and communication mismatch. This is an issue because in practice this calculus is supposed to be a helpful, concise abstraction of the code implementation, that helps focus on the communication aspect and reduce mistakes. However while it helps with formalised description, it is not sufficient to guarantee correctness properties. For this we can use session types.

2.3 Session Types

As mentioned before the objective of session types is to offer the same type of support for concurrent communication protocols that normal types and data structures provide in typical programming languages. Hence they aim to act as units of compositionality, and to guide developers in their implementation efforts [3]. Formally session types perform syntactic local checks at compile time guaranteeing that the following properties hold at run time [6]:

1. **Communication safety**: no mismatch between the types of sent and expected messages.

2. **Protocol fidelity**: correspondence between the interactions that occur at run time and those accounted for by the types and therefore allowed by the protocol.

3. **Progress**: deadlock guaranteed not to occur.

To illustrate how the type checks work, in Table 2.2 we provide the type syntax for the previously introduced calculus.

Using the example from Section 2.2.1 we explore the application of this typing syntax. We provide the type of the occurrences of session channel *v* in the two subprocesses representing the tool and the user. This is a syntactic operation that takes a π-calculus expression and step by step extrapolates the type using the rules provided above.

Alice: $?[\texttt{nat}]; ![\texttt{nat}]; \texttt{end}$
Bob: $![\texttt{nat}]; ?[\texttt{nat}]; \texttt{end}$

However being able to tell the local type of a subformula does not guarantee any property. The types of the two parallel subprocesses need to be compared to assert that their interaction is correct. In other words, we need to perform *type judgement*.

$S ::=$		***Sort***
	$\texttt{bool} \mid \texttt{nat} \mid \texttt{string}$	
$T ::=$		***Types***
	$![\tilde{S}];T$	*sending sort of type S with continuation with type T*
	$![T];T'$	*sending session of type T with continuation with type T'*
	$?[\tilde{S}];T$	*receiving sort of type S with continuation with type T*
	$\&\{l_1 : T_1, ..., l_n : T_n\}$	*branching*
	$\oplus\{l_1 : T_1, ..., l_n : T_n\}$	*selection (does not need all option in local context)*
	t	*recursion call*
	$\mu t.T$	*recursive behaviour*
	$\texttt{end}$	*end of session*

Table 2.2: Session types syntax

2.3.1 Typing judgement

The objective of type judgement is to assert that the occurrences of a channel in parallel processes are *dual*. In other words, it checks that channels are used in a way that allows valid interactions to occur and progress to be made.

Duality can be asserted syntactically using the following acceptable pairings:

$$\overline{![\tilde{S}];T} = ?[\tilde{S}];\overline{T} \quad \overline{![T];T'} = ?[T];\overline{T'} \quad \overline{\&\{l_i : T_i\}} = \oplus\{l_i : \overline{T_i}\}$$
$$\overline{?[\tilde{S}];T} = ![\tilde{S}];\overline{T} \quad \overline{?[T];T'} = ![T];\overline{T'} \quad \overline{\oplus\{l_i : T_i\}} = \&\{l_i : \overline{T_i}\}$$
$$\overline{\texttt{end}} = \texttt{end} \qquad \overline{\mu t.T} = \mu t.\overline{T} \qquad \overline{t} = t$$

In the current example the type checks pass because the interaction occurs correctly, progressing appropriately to the end of the communication.

$$\overline{?[\texttt{nat}];![\texttt{nat}];\texttt{end}} = ![\texttt{nat}];\overline{![\texttt{nat}];\texttt{end}} = ![\texttt{nat}];?[\texttt{nat}];\overline{\texttt{end}} = ![\texttt{nat}];?[\texttt{nat}].\texttt{end}$$

However, simple alterations to the protocol–formally representable with the calculus–could invalidate the interaction. Changes like sending a string instead of a number, or sending two values in a row instead of sending and waiting for a response in the user subprocess would make the reduction impossible. The types would not be dual and the type judgement would fail.

These are the basic concepts introduced with binary session types [12] [28] , and constitute the fundamentals necessary to understand this project.

These concepts extend to *multi-party session types* [6] [5] which makes these findings applicable to more complex systems. However in *multi-party* session types there

are two levels of typing abstraction. The higher level one is that of *global types*. These describe the protocol from a neutral standpoint essentially modelling the interactions between the peers, their order and the type of messages exchanged. At a lower level of abstraction *local types* describe the communication from the single peer viewpoint–specific actions that it will performs on messages and order.

This certainly complicates the model, but fortunately in practice type checks still rely on local types. This is due to the tight connection between global and local types. It is in fact possible to extract single local types from the global types, and vice-versa from local types to the global type. The former transformation is called projection and the latter synthesis. Conveniently this mapping causes checks on local types to give global guarantees. This is an extremely useful property as it allows the type-checker to focus on smaller sections of a large distributed system at a time. While initially all local types need to be checked, if any local component is then changed while the rest stays the same, only the local type of the modified component needs to be rechecked, which leads to better efficiency and performance.

We can now introduce the actual calculus behind the implementation of effpi, the Higher-order π-calculus.

2.4 Higher-Order π-calculus

$P, Q ::=$	
end	*Nil process*
$u?(x).P$	*input of x on channel u with continuation P (x scoped in P)*
$u!\langle v\rangle.P$	*output of v on channel u with continuation P*
let f = $\lambda\, x.P$ **in** Q	*abstraction of process f with parameter x*
$f\ u\ v$	*application of abstract process f taking values u and v, which can be normal types such as `String`, channels, or even abstract processes*

Table 2.3: Partial syntax of the HOπ

The correctness properties described in Section 2.3 are more difficult to check when dealing with higher-order concurrent programs that can exchange communication channels and mobile code. Firstly we need to extend the π-calculus syntax to be able to formally describe such systems. This can be done through the higher-order π-calculus (HOπ) [23], which allows us to express, on top of what is possible through the syntax from Section 2.2, the sending and receiving of processes.

This is done by introducing abstract processes, similar to an *abstraction* in λ-calculus. In practice, this allows us to define processes such as **let** $f = \lambda o.o!\langle 1\rangle$ where λo

indicates that the channel of the process is actually parametrised and that will be passed in an application of f with a concrete channel to take o's place.

While *abstraction* and *application* are the main additions to the syntax, there are also a few other syntactic differences, illustrated in Table 2.3.

2.4.1 Example V: Ping-Pong system

We can use this new syntax to represent a simplified version of the Ping-Pong system described in [13] and later used as a benchmark in Section 5.3, with a single pair of processes, doing a single ping-pong back and forth of messages with no recursion. *Ping* sends its channel to *Pong* to receive a response (we use $()$ to indicate a value of type `Unit`).

let $Pong = \lambda\ p_o.(p_o?(x).x!\langle()\rangle.\mathbf{end})$ **in** (

let $Ping = \lambda\ p_i.\lambda\ p_o.(p_o!\langle pi\rangle.p_i?(x).\mathbf{end})$ **in** (

$\nu\ p_i.\ \nu\ p_o.(Ping\ p_i\ p_o \mid Pong\ p_o)$

))

In Section 3.1 we present the Effpi implementation of thi system and we compare it to this this representation to assess the similarity and give an intuition of Effpi's grounding on this formal calculus.

2.5 Behavioural abstract process types for HOπ

While the syntax of HOπ allows us to have parametrised abstract processes, the session types described in Section 2.3 are *channel types*, which are only capable of describing the behaviour of a given channel.

What is introduced in [24] are *abstract process types* which describes in details the runtime behaviour of processes. It does so using dependent function types, which are of the form:

$$\Pi(in : \texttt{InputType})\texttt{FunctionReturnType}$$

which represents closures, with typed and named input which can be referenced in the `FunctionBodyType` thus allowing to detail the behaviour of the function within the type.

The full type definition can be found in [24], however in Table 2.5 we introduce the elements necessary to express the $Pong$ system from Section 2.4.1 and to give an intuition on the theory on which Effpi is grounded.

$S ::=$		***Sort***
	$\texttt{bool} \mid \texttt{nat} \mid \texttt{string} \mid \texttt{Unit}$	
$S, T, U ::=$		***Types***
	$\Pi(x : T)U$	*dependent function type*
	$o[S, T, U]$	*describes a process that sends a T-typed value on an S-typed channel, and continues as U*
	$i(S, T)$	*describes a process that receives a value from an S-typed channel and continues as T*
	$\texttt{nil}$	*unit type*

Table 2.5: Partial definition of behavioural types as presented in [24], upon which Effpi is grounded

Typing of the Pong Process with behavioural types

Using this type system, the type of the abstract process $Pong$ introduced in Section 2.4.1 is:

$$Pong : \Pi(po : Chan^I[Chan^O[\texttt{Unit}]])\ i(po,\ \Pi(x : Chan^O[\texttt{Unit}])\ o(x, \texttt{Unit}, \texttt{nil}))$$

Breaking this type down into its component helps understanding how the type is a detailed description of the process behaviour:

- $\Pi(po : Chan^I[Chan^O[\texttt{Unit}]])...$: we can see that $Pong$ first takes an input channel po of type $Chan^I[Chan^O[\texttt{Unit}]]$, which is an input channel that yields an output channel, that in turn allows to send a message of type `Unit`(the type of $()$). The return type of this expression will be able to reference po directly.
- $i(po,\ \Pi(x : Chan^O[\texttt{Unit}])\ ...)$: This is the first part of the return type, which indicates that an input x of type $Chan^O[\texttt{Unit}]$ has to be received through channel po. This is possible because the type of the value matches the type that was given to channel po in the input. The reference to po means that any value of type $Chan^I[Chan^O[\texttt{Unit}]]$ would not be sufficient. It requires a value of the type of po, which is only satisfied by po itself, thus ensuring the correct use of the channel.
- $o(x, \texttt{Unit}, ...)$: this means that a value of type `Unit` is sent through channel x. Once again this is possible because this checks with the type just defined for x in the previous step.

- `nil`: the process terminates.

Since these types allow for a very detailed description of the processes behaviour, their correctness can be established through type judgement (beyond the scope of this overview, but is explained in details in [24]). In Section 3.1 this types are compared to those implemented in Effpi to provide an intuition of their strict relationship that allows Effpi to provide the same correctness guarantees.

Chapter 3

Background: Effpi - an overview

In this chapter we explore the architecture of the Effpi library, its DSLs and the types provided to perform correctness checks. We then analyse the original implementation of the runtime to understand its flaws. This should help contextualise the contributions made in later chapters, highlighting the reason for their necessity and giving a background that should ease the understanding of their practical implementation.

3.1 An introduction to Effpi

As mentioned in the introduction Effpi provides an API composed of two DSLs, one based on π-calculus and one based on the actor model, that helps writing concurrent programs receiving strong guarantees of correctness through type-checking. This is done through deep embedding of the DSLs [27] in Scala. More specifically, the DSLs help build data structures representing the behaviour of the program being developed. These data structures are strongly typed and their types therefore also reflects the program behaviour. This allows type-checking to ensure that a program behaves as described by its type.

Before we dive into Effpi's details, in Listing 3.1 we provide an Effpi implementation of the Ping-Pong system described in Section 2.4.1, to show how the DSL closely matches the HOπ representation. Furthermore Effpi's types are clearly resemblant of the theoretical types introduced in Section 2.5, providing a similar detailed description of the process behaviour. This only aims to provide an intuition of this correlation, a more detailed explanation is provided in [24].

Listing 3.1: Effpi implementation of a simple PingPong system

```
type TPong = (
  (po: InChannel[OutChannel[Unit]]) =>
    In(po.type, (x: OutChannel[Unit]) =>
      Out(x.type, Unit, (Unit) => Nil) ))

def pong(po: InChannel[OutChannel[Unit]]): TPong = {
  receive(po) { x =>
    send(x, ()) {
      nil
    }
  }
}

type TPing = ... // Omitted

def ping(pi: Channel[Unit], po: OutChannel[OutChannel[Unit]]): TPing = {
  send(po, pi) {
    receive(pi) { x =>
  }
}
```

While the project is mostly concerned with improving Effpi with a new improved runtime, it is necessary to have a general understanding of how the other parts of the system work. In particular the data structures produced by the Process DSL are directly parsed and processed by the runtime, and therefore it is necessary to understand how the DSL is used to create them in order to appreciate this project's contributions. It is also useful to have familiarity with the Actor DSL as it provides a simpler API, and because all the benchmarks and examples are implemented using it (including the Ping-Pong system from Listing 3.1). However there is also a number of other internals that make up the architecture and there is a dependency structure between the different concepts. It is therefore useful to explore these in order to understand their purpose, which is why we will start describing the concept of Channel.

3.2 Channels

The concept of Channel is fundamental in π-calculus, and therefore also in Effpi. All other concepts depend on it and therefore it is a good place to start.

The main two types of channels in Effpi are InChannels and and OutChannels, described in Effpi by the homonym traits shown in Listing 3.2. These describe two fundamental behaviours: the ability to receive and the ability to send.

Listing 3.2: Original `InChannel` and `OutChannel` traits

```
trait InChannel[+A] {
  val in: InChannel[A] = this
  def receive()(implicit timeout: Duration): A
}

trait OutChannel[-A] {
  val out: OutChannel[A] = this
  def send(v: A): Unit
}
```

In practice the implementation of sending and receiving adopted by both DSLs, used in the system and in the runtime, is given by the traits `QueueInChannel` and `QueueOutChannel` (Listing 3.3, Listing 3.4). It is important to notice that the implementation of `receive` is blocking, and can be given a maximum time out duration, after which a runtime error is thrown.

Listing 3.3: Original implementation of `receive` through `QueueInChannel` trait

```
trait QueueInChannel[+A](q: LTQueue[A]) extends InChannel[A] {

  override def receive()(implicit timeout: Duration) = {
    if (!timeout.isFinite) {
      q.take()
    } else {
      val ret = q.poll(timeout.length, timeout.unit)
      if (ret == null) {
        throw new RuntimeException(s"${this}: timeout after ${timeout}")
      }
      ret
    }
  }
}
```

Listing 3.4: Original implementation of `send` through `QueueOutChannel` trait

```
trait QueueOutChannel[-A](q: LTQueue[A])(maybeDual:
    Option[QueueInChannel[Any]]) extends OutChannel[A] {
  override def send(v: A) = q.add(v)
}
```

3.3 Processes and the process DSL

The next important concept is that of process. As seen in the background, in π-calculus there are many different types of processes, depending on the syntax, which allow for sending, receiving, creating two parallel sub-processes and so on.

These are expressed in Effpi as case class implementation of the sealed abstract class Process (Listing 3.5). This is a common pattern in Scala when there is a sealed abstract class or trait implemented by a number of case classes. This means that pattern matching can be done on the sealed entity and that if all the implementing cases are considered the pattern matching will be seen as exhaustive by the type checker. Any missing case on the other hand will be spotted. This implementation comes useful in the runtime, where the correct processing of each possible process type is extremely important.

The Process class also implements the `spawn` function that takes care of instantiating a concurrent computation that performs the process. This logic is part of what constitutes the original runtime and is explained in more details later in Section 3.5.

Listing 3.5: Original implementation `Process`

```
1  sealed abstract class Process {
2    def >>[P1 >: this.type <: Process, P2 <: Process](cont: => P2) = >>:[P1,
         P2](() => this, () => cont)
3
4    def spawn() = {
5      val self = this
6      val t = new Thread { override def run = eval(Map(), Nil, self) }
7      t.start()
8      t
9    }
10 }
```

The implementations of Process are reported in Listing 3.6. It is important to notice that each of this classes has a specific type, and that these type-checking these classes is what provides guarantees about the programs behaviour.

It is also important to point out that there is a strong correspondence between the types of these elements and those described in [24]. One notable difference is that *send* in [24] has a direct continuation, while in this DSL this is not the case, and by default the process ends after an `Out`. However, for convenience, the DSL offers the sequence building block, $>>:$, which allows us to manually add continuation. This is implemented by the $>>$ operator of `Process` (Listing 3.5).

Listing 3.6: The different types of processes are expressed as implementations of `Process`

```
case class Out[C <: OutChannel[A], A](channel: C, v: A) extends Process

case class In[C <: InChannel[A], A, P <: Process](
  channel: C, cont: A => P, timeout: Duration
) extends Process

case class Fork[P <: Process](p: () => P) extends Process

sealed abstract class PNil() extends Process

case class Def[V <: ProcVar, A, P1 <: Process, P2 <: Process](
  name: V[A], pdef: A => P1, in: () => P2
) extends Process

case class Call[V <: ProcVar, A](procvar: V[A], arg: A) extends Process

case class >>:[P1 <: Process, P2 <: Process](
  p1: () => P1, p2: () => P2
) extends Process
```

3.3.1 DSL

While the implementations of Process are the actual building blocks that are used to create a data structure that describes in details a process, these are not to be used directly by the user for the construction of a process. While possible to do, this would not be neither convenient nor practical. This is the reason behind having the the process DSL as a public API. We describe now the main functions it provides, and compare them to the syntax of the calculus from [24].

Sending: `send` corresponds to **send** from the calculus. The function passes the channel and the value to build an `Out` block.

Listing 3.7: Process DSL: send

```
def send[C <: OutChannel[A], A](c: C, v: A) = Out[C,A](c, v)
```

Receiving: `receive` corresponds to **recv** as it takes the input channel and the continuation as arguments. However, since the implementation of `InChannel` this is based on requires a time-out to handle waiting on the read, this is also taken as an

implicit parameter. The advantage of `timeout` being implicit is that the caller code can be cleaner.

Listing 3.8: Process DSL: receive

```
def receive[C <: InChannel[A], A, P <: Process]
(c: C)(cont: A => P)(implicit timeout: Duration) = In[C,A,P](c, cont,
    timeout)
```

Creating parallel processes: `fork` corresponds to the parallel composition of two processes **P** | **Q** for processes **P** and **Q**

Listing 3.9: Process DSL: forking parallel processes

```
def fork[P <: Process](p: => P) = Fork[P](() => p)
```

Creating abstract processes: `pdef` vaguely corresponds to **let** $x = t$ **in** t'. This is because that syntax can be used for any sort of definition, while this DSL function (and the `Def` structure itself) is specific for abstract processes. The definition of normal variables, for example, can be done with plain Scala. The arguments `name`, `pdef` and `in` correspond respectively to x, t and t' the first two establishing a mapping between a name and the body of the definition and the last being the continuation that is also the scope of the definition just established.

Listing 3.10: Process DSL: abstract process definition

```
def pdef[V <: ProcVar, A, P1 <: Process, P2 <: Process]
(name: V[A])(pdef: A => P1)(in: => P2) = Def[V, A, P1, P2](name, pdef, ()
    => in)
```

Applying abstract processes definitions: `pcall` corresponds to application $t\ t'$, with `name` corresponding to t and `arg` to t'.

Listing 3.11: Process DSL: calling abstract processes definitions

```
def pcall[V <: ProcVar, A](name: V[A], arg: A) = Call[V,A](name, arg)
```

Recursion: `rec` is a convenience method based on `Def` that simplifies the description of recursive processes. This is used in the benchmarks, as most large benchmark require some sort of recursive behaviour, as there is no concept of looping.

Listing 3.12: Process DSL: calling abstract processes definitions

```
def rec[V <: RecVar, P <: Process](v: V[Unit])(p: => P): Rec[V, P] =
  Def[V, Unit, P, P](v, (x) => p, () => p)
```

Remarks:

The π-calculus syntax provides and `if-then-else` construct, however this is already provided by Scala, and therefore it is not necessary to reproduce it in the DSL. The operator $>>$ from Listing 3.5 is also integral part of the DSL, as it allows us to have continuation on `send`, which should be possible by the calculus described in [24].

3.4 Actors and Actor DSL

The Actor DSL is somewhat of an extra, experimental feature of Effpi. The main goal was to have a pi-calculus based concurrency framework, however due to the popularity of actor base frameworks, providing an actor based API could facilitate the framework's adoption.

The Actor DSL is actually implemented using the Process DSL under the hood, which is why it provides the same correctness guarantees. It actually serves as a façade to the Process DSL, providing a simpler API, which also allows us to enforce certain properties beneficial to actor programming (for example enforcing that each actor has only one mailbox). However in order to provide this added simplicity some Actor specific internals had to be introduced.

3.4.1 Mailbox & ActorRef

Given that the core characteristic of the actor model is to perform asynchronous computation through message passing, and since actors are the primary concept in this model, they must be able to send and receive messages. An actor Mailbox is what allows an actor to receive messages–that is an actor can check its mailbox for incoming messages from other actors. ActorRefs conversely are addresses can be used to send actors messages.

In Effpi Mailboxes and ActorRefs are internally implemented extending the `InChannel` and `OutChannel` traits (Listing 3.13). The only addition is to the ActorRef interface, which is provided with a `!` operator,

Listing 3.13: Traits for Mailboxes and ActorRefs to use in the Actor DSL

```
abstract class Mailbox[+A] extends InChannel[A]

abstract class ActorRef[-A] extends OutChannel[A] {
   def ! = send
}
```

3.4.2 Behaviours and Actor Context

In order to encourage good patterns by design there are a few other concepts implemented to allow the Actor DSL to spawn actors with their unique Mailbox and ActorRef accessible within their context.

A fundamental construct is the `Behavior` class (Listing 3.14). Essentially a behaviour allows for the creation unexecuted closures. This closures represent the processes defined using the DSL.

Listing 3.14: Definition of `Behavior` to create unexecuted closures representing the processes

```
  type WithMailbox[A, T] = implicit ActorCtx[A] => T

  class Behavior[A, P](body: => WithMailbox[A, P])
        extends Function0[WithMailbox[A, P]] {
    def apply() = body
  }
```

The type of the closures, `WithMailbox`, ensures that they exist in an Actor Context (Listing 3.15), meaning they have access to an ActorRef and a Mailbox implicitly.

Listing 3.15: Actor Context, used to ensure a mailbox is implicitly available

```
abstract class ActorCtx[A](val self: ActorRef[A],
                           protected[actor] val mbox: Mailbox[A])
```

So checking the types of these unexecuted closures is actually what guarantees the correctness of the behaviour of the processes created. The runtime can subsequently execute the processes by making calls to the closures.

One last important detail is that the Actor DSL relies on actors being spawned using the `Actor.spawn` function (Listing 3.16). This wraps the `spawn` function from `Process` introduced in Listing 3.5, but at the same time establishes the ActorRef and Mailbox to be used for the actor and to be passed to the Actor Context.

Listing 3.16: The `Actor.spawn` function creates an actor providing it an Actor Context

```
object Actor {
  def spawn[A, P <: Process](beh: Behavior[A, P])(implicit ps:
      ProcessSystem): ActorRef[A] = {
    val pipe = ActorPipe[A]()
    implicit val ctx = new ActorCtxImpl[A](pipe.ref, pipe.mbox)
    val proc = beh()
    proc.spawn(ps)
    pipe.ref
  }
}
```

3.4.3 The Actor DSL

While the Actor DSL provides a simpler API than the Process DSL, most functions needed to understand the benchmarks are actually fairly similar to those presented for the Process DSL, and internally depend on them anyway. Just as an example we show how this is the case for sending and receiving in Listing 3.17. Note that `psend` is just an alias for the send function of the Process DSL

Listing 3.17: The `read` and `send` functions of the ActorDSL

```
def read[T, P <: Process]
  (cont: T => P)
  (implicit timeout: Duration): WithMailbox[T, Read[T, P]] = {

  preceive[Mailbox[T], T, P](implicitly[ActorCtx[T]].mbox)(cont)(timeout)
}

def send[R <: ActorRef[T], T](ref: R, x: T): SendTo[R, T] = psend(ref, x)
```

The main advantage in using the Actor DSL in general is that it provides channels implicitly as explained in the previous section. In Listing 3.18 we provide an example of how this leads to more readable code, by implementing the same Ping-Pong system from Listing 3.1 using the Actor DSL.

Intuitively we can see that the actor-style `PongProcess` type performs a receive action of a value of type `Ping`, and then continues with a send action of a value of type `Pong` through a channel that accepts `Pong`-typed messages. We can notice that the main difference between this definition and the one given in Listing 3.18 (and in turn that from Section 2.5) is that in the actor-style code the input channel from which the pong process receives the Ping-typed value is kept implicit, and does not appear as a

parameter in `Read`. This reminds the implicit actor mailboxes used in Erlang or Akka programs.

Listing 3.18: Effpi implementation of a simple PingPong system

```
final case class Ping(replyTo: ActorRef[PongMessage])
final case class Pong()

type PongProcess = Read[Ping, SendTo[ActorRef[Pong], Pong]]

def pong = Behavior[Ping, PongProcess] {
  read {
    case Ping(replyTo) =>
      send(replyTo, Pong())
  }
}

type PingProcess[R <: ActorRef[Ping]] = SendTo[R, Ping] >>: Read[Pong,
    PNil]}

def ping(pongRef: ActorRef[Ping]) = Behavior[PongMessage,
    PingProcess[pongRef.type]] {
  send(pongRef, Ping(self)) >>
    read {
      case PongMessage.Pong =>
        nil
    }
}
```

Furthermore it is important to notice that, as long as the function names are disambiguated, the two DSLs can actually be mixed seamlessly given that the Actor DSL is simply a wrapper. This property is used in the benchmarks where the simpler Actor DSL is preferred, however the `rec` function from the Process DSL is used to allow for recursive behaviour which helps describing larger and more complex systems (see for example the Chameneos benchmark in Appendix A).

3.5 Original run-time

The original runtime was intended to provide the bare minimum functionality that would allow the framework to work. As such it follows the naive strategy of spawning a JVM thread per process to guarantee their concurrent execution. Going into the details of this implementation helps highlight its limitations and will be useful to understand the changes made.

The two main concepts behind the runtime to understand are the mechanism to maintain state and the actual strategy for the execution. The former is required as it provides a sort of execution context, that at all times contains the information needed to perform correctly the behaviour of a process as it is consumed. The latter contains the actual logic that tells what actions should be taken for each process type, meaning what to do for each kind of process (`read`, `write`, `fork` and so on...)

Execution context

It is important to establish what the execution context is in this framework. In other words, what kind of state is needed to be able to execute a process. The execution context should store enough information to maintain the state of a process as the evaluation unravels the intricate and recursive `Process` data structure. As layers are evaluated some of the information that might otherwise be lost must be stored in a convenient fashion to be available to inner parts of the `Process`.

The **process state**, as it will be referred to moving forward, consists of the following three pieces of information:

- **current process:** The process the evaluation function is currently consuming. This is the main focus of the evaluation and when the process execution just starts it is also the only information provided.
- **future processes:** This is a list containing processes that must still be evaluated when the current process finishes (becomes `nil`). It can be thought of a 'secondary' continuation. This is due to types like `Out` not having a continuation embedded in their type and relying on the $>>:$ to have one. In these cases the evaluation continues on the current process and the continuation is appended onto this list.
- **environment:** The last element of the triplet is a map storing the process definitions encountered so far. This is needed to deal with the `Def` and `Call` process types. This map allows to keep definitions in memory where they are available in the needed scope, and to access them efficiently.

Run strategy

The way a system of processes runs can be conceptually divided into three steps: process construction, spawning, evaluation.

Firstly a process is constructed using the DSL, which creates a data structure that works as a definition of the process behaviour.

Then the method `spawn` can be called (Listing 3.5). This method has the responsibility to create a JVM thread inside which the process can be executed concurrently, through the `eval` function. When using the Actor DSL one can use the `Actor.spawn` function (Listing 3.16), which has the advantage of implicitly setting up a Mailbox and an ActorRef.

`eval` is where the logic for the evaluation resides, and as such defines what is meant by *executing a process*. This function is tail recursive, reflecting the recursive nature of the `Process` data structure. At each call it unravels one layer of the process, defined as one step of the process execution. It then generally recurs on the continuation of the process. Naturally since all the logic for the execution is contained in this function, it takes the triplet representing the process state as argument (`(env, lp, p)`), to guarantee the necessary knowledge to perform the evaluation is always available. Diving into the implementation of `eval`, the first important task it performs, shown in Listing 3.19, is to pattern match on the current process `p`. This is necessary because different actions are required for each process type in order for it to reflect its semantic meaning.

Listing 3.19: The overall structure of the `eval` function

```
def eval(p: Process): Try[Unit] = Try(eval(Map(), Nil, p))

@annotation.tailrec
def eval(env: Map[ProcVar[_], (_) => Process], lp: List[() => Process],
        p: Process): Unit = p match {
  case i: In[_,_,_] => ???
  case o: Out[_,_] => ???
  case f: Fork[_] => ???
  case n: PNil => ???
  case d: Def[_,_,_,_] => ???
  case c: Call[_,_] => ???
  case s: >>:[_,_] => ???
}
```

Then in general the pattern for most types of processes is:

- evaluate the current step of the process and perform the required actions
- make a recursive call to `eval` passing as argument a process state that looks like `(env, lp, p.con)`, where by `p.con` we mean the continuation of the current process `p`

However this is not true for all cases, and the action taken for the specific step differs from type to type.

PNil

This is somewhat of a special case, because `nil` represents the **0** process and therefore it does not have a direct continuation.

However the list of future processes might provide a continuation, and therefore `lp` is pattern matched and if it is not empty then the head of the list becomes the new current process for the state to be passed as an argument to a recursive call to `eval`, taking the form `eval(env, tail(lp), head(lp))` However if `lp` is empty the process is truly over, and the evaluation terminates.

Listing 3.20: `eval` evaluation of the `PNil` case

```
case n: PNil => lp match {
  case Nil => ()
  case lh :: lt => eval(env, lt, lh())
}
```

In

In case of `In` a value is read from the InChannel accessible as a field of the process. This is done through the channel's `receive` method, which blocks and waits for a value to be passed. However a time out is set on how long this wait can last (Listing 3.3), and if it is not successful by then a runtime exception will be thrown. This blocking implementation makes sense, because since each process is assigned a JVM threads these will handle context switching to allow other processes to run. At the same time the time out provides a sanity check to the status of each input-process.

If the read is successful, however, a recursive call to `eval` is made, with the state of the continuation as argument. Said continuation of the process `i` is obtained through the field `cont`.

Listing 3.21: `eval` evaluation of the `In` case

```
case i: In[_,_,_] => {
  val v = i.channel.receive()(i.timeout)
  eval(env, lp, i.cont.asInstanceOf[Any => Process](v))
}
```

Out

In `Out` the first operation is to send a value to the OutChannel accessible as a field of the current process `o`.

As we saw in Section 3.3, `Out` does not have a direct continuation like `In`. Therefore the list of future processes is pattern matched. If the list is empty the process is over and the execution terminates, but if it is not then a recursive call to `eval` is made in a similar fashion to the `PNil` case.

Listing 3.22: `eval` evaluation of the `Out` case

```
case o: Out[_,_] => {
  o.channel.asInstanceOf[OutChannel[Any]].send(o.v)
  lp match {
    case Nil => ()
    case lh :: lt => eval(env, lt, lh())
  }
}
```

Fork

Fork is supposed to allow for parallel processes, embodying the syntax $P|Q$ for processes P and Q. For this reason it spawns a new thread running another instance of `eval`. This takes a process state that inherits the environment and takes an empty list as future processes and the parallel process `p` specified as a field of the current process `f`. A `Fork` type process does not have a direct continuation, so it continues the execution on the current thread with a behaviour similar to that of `PNil` and `Out`, matching on the list of future processes `lp`.

Listing 3.23: `eval` evaluation of the `Fork` case

```
case f: Fork[_] => {
  new Thread { override def run = eval(env, Nil, f.p()) }.start()
  lp match {
    case Nil => ()
    case lh :: lt => eval(env, lt, lh())
  }
}
```

Def

The `Def` process is supposed to establish a definition of a process assigning it a name. As a result the only action taken prior to making a recursive call, is to add a definition to the environment of the state, now updated with one more mapping from `d.name` to `d.pdef`. It also sets the current process to the process stored in the `in` field of `d`. While theoretically it could have continued with `lp` like in `Fork`, this design was chosen to make it more similar to the syntax from [24], with the `in` process representing the scope of the definition.

Listing 3.24: `eval` evaluation of the `Def` case

```
case d: Def[_,_,_,_] => {
  eval(env + (d.name -> d.pdef), lp, d.in())
}
```

Call

`Call` tries to access a definition in the environment to use it in the continuation. This could potentially cause an error, if the definition needed is not found. In this case a runtime exception is thrown.

Listing 3.25: `eval` evaluation of the `Call` case

```
case c: Call[_,_] => {
  env.get(c.procvar) match {
    case Some(p) => {
      // println(s"*** Calling ${c.procvar}(${c.arg})")
      eval(env, lp, p.asInstanceOf[Any => Process](c.arg))
    }
    case None => {
      throw new RuntimeException(s"Unbound process variable: ${c.procvar}")
    }
  }
```

>>:

For the sequence operator the implementation is quite straightforward: take `s.p1()` as continuation and append `s.p2` at the front of the future processes list.

Listing 3.26: `eval` evaluation of the >>: case

```
case s: >>:[_,_] => {
  eval(env, s.p2 :: lp, s.p1())
}
```

3.5.1 Final Remarks

As described in the Introduction (Section 1.1), the problem with this implementation is that not decoupling effpi-processes from JVM threads it only supports a very limited number of processes before the program becomes non-responsive.

Furthermore, once the number of threads greatly exceeds the cores available to the host hardware, this does not bring any significant advantage since the parallelism will still be limited. Improving on this limitation is the focus of the next chapter.

Chapter 4

Design and Implementation

4.1 Design overview

This chapter presents a number of optimised implementations of the runtime that address the objectives described in the Introduction (Section 1.2). The first fundamental goal of this project was to allow Effpi to run large concurrent systems with hundreds of thousands of processes without the program becoming non-responsive, and the first implementation (referred to as RunningQueue) described in Section 4.2 brings improvements on this aspect. Another important objective was that of speed of execution, that is minimising the bookkeeping operations performed during scheduling to maximise the time spent performing actions that bring the execution forward. This issue is addressed by the optimised implementations WaitQueue (Section 4.3), WaitQueueImproved (Section 4.4) and MultiStep (Section 4.5).

An important concept that helps understand these implementations is that of Execution Context of a process, introduced in Section 3.5. This represents all the information required to successfully execute a process, and is represented by the triplet `(env, lp, p)`, where `env` is the mapping storing all the process definitions, `lp` is a list of future continuations, and `p` represents the current process. The common characteristic of all following runtime implementations is that in order to decouple processes from JVM threads they store the processes to run in memory and only spawn a limited number of threads. The processes in memory are then gradually consumed and executed by the threads. All this is managed by a structure called Process System shown in Figure 4.1. This should reduce the overhead due to creating too many JVM thread, which is the core reason why all the following implementations can sustain much larger systems than the original runtime.

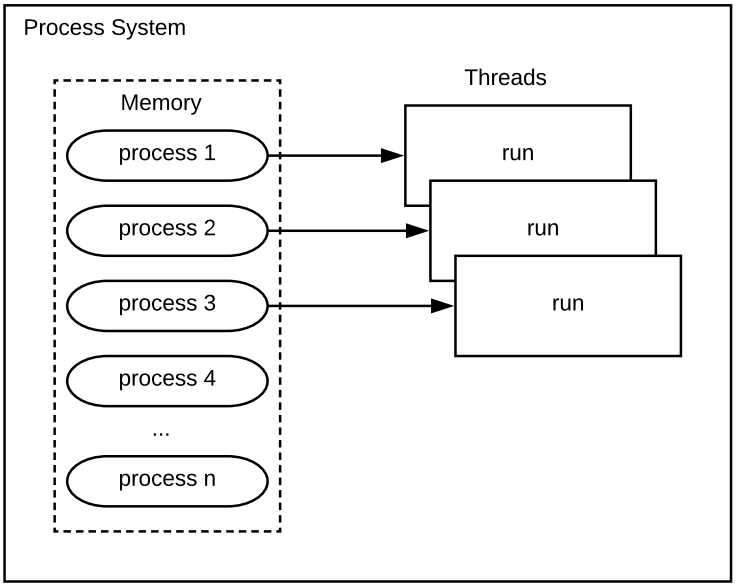


Figure 4.1: General architecture of a process system. With different practical implementations, this kind of structure will be used in all following implementations

4.2 Running Queue Implementation

4.2.1 Process System

The main concept behind this implementation, pictured in Figure 4.2, is that already mentioned of process system. The `ProcessSystem` object has two practical responsibilities in order to guarantee efficient memory based execution:

- Store in memory the state of all running processes.
- Gradually execute such processes by scheduling them on a limited number of JVM threads (the exact number of threads spawned is later discussed in the Evaluation, Section 5.4).

Running Queue

`ProcessSystem` stores a queue of states of active processes that are waiting to be executed. This leads to a queue of the type presented in line 3, that contains process state triplets (described in 3.5) which contains:

- the environment (a map storing the definition of processes)

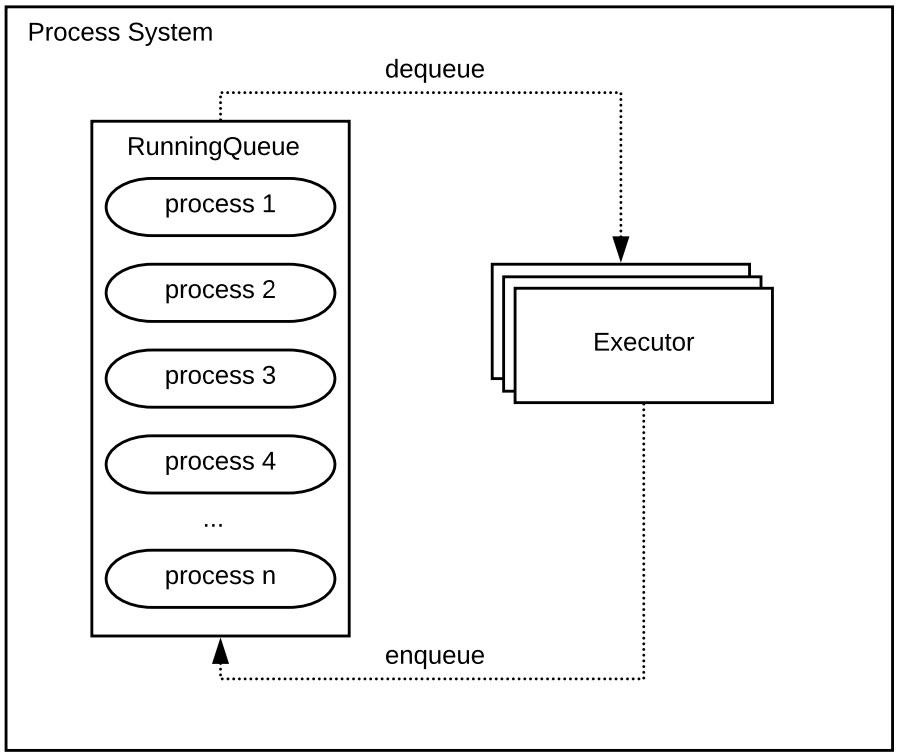


Figure 4.2: Architectural diagram of the RunningQueue implementation. The executors here consume and reschedule processes from a queue in memory

- the list of future processes to be evaluated
- the current process to evaluate.

As part of the `ProcessSystem` trait the methods `enqueueRunning` and `dequeueRunning` are also provided for consuming elements of that queue.

Listing 4.1: `ProcessSystem` storing a Running Queue

```
1  trait ProcessSystem {
2
3    var runningQueue = new SchedulingQueue[(Map[ProcVar[_], (_) => Process],
         List[() => Process], Process)]
4
5    def enqueueRunning = runningQueue.enqueue
6    def dequeueRunning() = runningQueue.dequeue()
7  }
```

Internally this queue is implemented using Java's `LinkedTransferQueues`, which are a well established and performant thread safe queue. This is important because the

Working Queue will be accessed concurrently by the threads in order to run execute the processes (Listing 4.2).

The important detail to notice about this queue is that although in theory it returns an Option, it is actually blocking, and therefore it will normally always return a value. The reason why an Option is returned is described in details in Section 4.2.6.

Listing 4.2: This is a wrapper around a Java `LinkedTransferQueue` which adds some logic on dequeueing needed for termination

```
class SchedulingQueue[E] {

  val queue = new LTQueue[E]

  def enqueue(elem: E) = {
    queue.add(elem)
    ()
  }

  def dequeue(): Option[E] = {
    try { Some(queue.take()) } catch {
        case e: InterruptedException =>
          None
      }
  }

}
```

Thread spawning

In order to consume the processes stored on the RunningQueue the Process System object is also responsible for assessing how many cores the host machine has and for making a decision about how many JVM threads to create [1].

This is all handled by the `init` function (Listing 4.3), which is called immediately after construction (construction here is performed using the apply method of the companion object, a fairly common and idiomatic pattern in Scala, which allows construction to be performed without the `new` keyword, like so `val ps = ProcessSystemRunningQueue()`.

[1] The current number has been established heuristically and is discussed further in the Evaluation chapter 5.4

Listing 4.3: Process System logic to spawn Executors in a limited number of threads

```
override def init(threadsPerCore: Int): Unit = {
  val numCores = Runtime.getRuntime().availableProcessors()

  val numThreads = numCores * threadsPerCore

  (1 to numThreads).foreach { c =>
    threads += new Thread(new Executor(this))
  }

  threads.foreach{ t => t.start()}
}
```

Each JVM thread runs an `Executor`, which is the class responsible for dealing with the evaluation of processes.

4.2.2 Evaluation with Executors

RunningQueue consumption

The key function in `Executor` which contains the logic for evaluating a process is `fastEval`. This is conceptually similar to the original `eval` described in Section 3.5, and in fact it has two significant differences:

- It is not exclusively bound to a single process and as a result, subsequent calls to `fastEval` may execute different processes.
- It is one-step, meaning each call to the function performed one step of the original recursive `eval` function (with one notable exception discussed later).

Listing 4.4: `Executor` logic to consume the Running Queue

```
override def run() = {
  while (ps.alive) {
    val head = ps.dequeueRunning()
    head match {
      case Some(p) =>
        fastEval(p)
      case None =>
        ()
    }
  }
}
```

Since `fastEval` performs one-step evaluation, it is run in an while loop to ensure it keeps consuming processes in the RunningQueue, where at each iteration a new process state is dequeued and evaluated (Listing 4.4).

fastEval implementation

Even though the signature of `fastEval` slightly differs from that of the original naive `eval`, accepting a tuple as an argument as opposed to three separate arguments, due to their conceptual similarity the substance of its implementation is left fairly unchanged. The first important task it performs is still to pattern match on the current process `p`, as shown in Listing 4.5. Moreover, the general pattern for most types of processes is only slightly changed. `fastEval` still evaluates the current step of the process and performs the required actions, but then instead of making a recursive call with the process state of the continuation, it simply enqueues this state back onto the Running Queue of the Process System.

Listing 4.5: Overview of `fastEval`

```
1  def fastEval(
2    proc: (Map[ProcVar[_], (_) => Process], List[() => Process], Process)
3  ): Unit = {
4    val (env, lp, p) = proc
5    p match {
6      case i: In[_,_,_] => ???
7      case o: Out[_,_] => ???
8      case f: Fork[_] => ???
9      case n: PNil => ???
10     case d: Def[_,_,_,_] => ???
11     case c: Call[_,_] => ???
12     case s: >>:[_,_] => ???
13   }
14 }
```

In and changes to InChannel

This new scheduling system requires one further change in order to be able to execute input-processes. Now that the one-to-one correspondence between processes and JVM threads is lost, it is no longer possible to wait on a read action for the entirety of the time-out. If all threads end up waiting on an input process and there is no way to continue the execution of other processes this might deadlock the execution. Instead we must have a fail-fast method to check if the channel has received any value, and if not simply re-enqueue the process back onto the `RunningQueue`.

Since the `receive` method of `InChannel` did not match the fail-fast requirement, a new non-blocking `poll` method is added to the trait, shown in Listing 4.6

Listing 4.6: `InChannel` trait updated with a non-blocking `poll` method

```
trait InChannel[+A] {
  val in: InChannel[A] = this
  def receive()(implicit timeout: Duration): A
  def poll(): Option[A]
}
```

However if the call to receive happens successfully, then the value is read and the continuation of process `p` can be considered in the new current process to enqueue, `(env, lp, cont)`. Notice that now an `In` process is not necessarily evaluated exclusively by a single JVM thread, and therefore we lose the possibility to use time outs naively as we did with the original runtime. This is discussed further in Future Work (Section 6.2).

Listing 4.7: `RunningQueue` implementation of `fastEval` in the `In` case

```
    case i: In[_,_,_] =>
      i.channel.poll() match {
        case Some(v) =>
          val cont = i.cont.asInstanceOf[Any => Process](v)
          ps.enqueueRunning((env, lp, cont))
        case None =>
          ps.enqueueRunning((env, lp, p))
      }
```

Out

The implementation of out remains fairly similar to that of the original eval. However after the value has been sent, if pattern matching of future processes `lp` reveals that the list is not empty, instead of having a recursive call, the continuation process of the form `(env, tail(lp), head(lp))` is enqueued onto the `Running Queue`.

Listing 4.8: `RunningQueue` implementation of `fastEval` in the `Out` case

```
    case o: Out[_,_] =>
      val outCh = o.channel.asInstanceOf[OutChannel[Any]]
      outCh.send(o.v)
      lp match {
        case Nil => ()
        case lh :: lt => ps.enqueueRunning((env, lt, lh()))
      }
```

Fork

As mentioned before this is the equivalent of $P|Q$, so it creates two parallel processes

It can be seen in Listing 4.9, the current implementation, instead of spawning a JVM thread for `f.p()` as in the naive `eval`, two new processes are enqueued. One being `(env, Nil, f.p())` and the other is the continuation of the parent, which consists in `(env, tail(lp), head(lp)`

Listing 4.9: `RunningQueue` implementation of `fastEval` in the `In` case

```
1   case f: Fork[_] =>
2     ps.enqueueRunning((env, Nil, f.p()))
3     lp match {
4       case Nil => ()
5       case lh :: lt => ps.enqueueRunning((env, lt, lh()))
6     }
```

The other cases

The other cases are left almost unchanged. The only difference with `eval` is that instead of making a recursive call the state of the continuation process is re-enqueued to the `Running Queue`. This is an illustration of this change for the `PNil` case:

Listing 4.10: `RunningQueue` implementation of `fastEval` in the `PNil` case

```
1   case n: PNil => lp match {
2     case Nil => ()
3     case lh :: lt => ps.enqueueRunning((env, lt, lh()))
4   }
```

4.2.3 Spawning effpi-processes

One thing left to complete the runtime is the ability to spawn processes. The old `spawn` function was the one responsible for creating the thread in which to run `eval`. In this implementation `spawn` is still a method of `Process`, but the implementation is changed. Now it is responsible to kick-start the consumption of the Running Queue by initially populating it, by enqueuing the process onto it, with an empty definition map and an initially empty list of future processes.

Listing 4.11: The `spawn` function used in all of the runtimes introduced in this project

```
1  def spawn(ps: ProcessSystem) = {
2    ps.enqueueRunning((Map(), Nil, this))
3  }
```

4.2.4 RunningQueue strategy on simple Ping-Pong example

In order to illustrate the details explained so far of the RunningQueue strategy for the runtime, we will analyse potential execution traces of a Ping-Pong concurrent system. We do so without explicit assumptions on the hardware, just that it can perform parallel execution. This is the same system detailed in Listing 3.1, with a single pair of processes, doing a single ping-pong back and forth of messages. Figure 4.3 illustrates its execution.

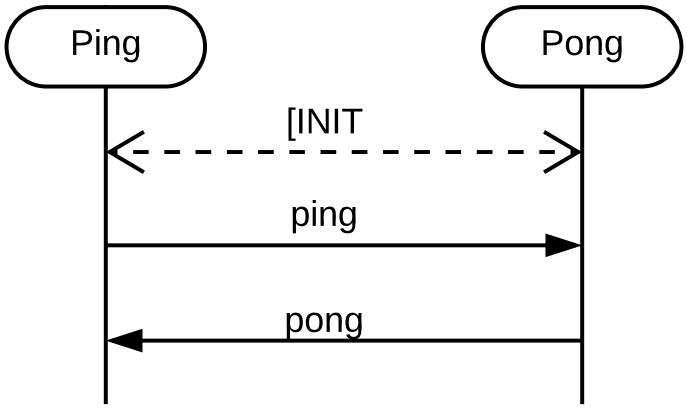
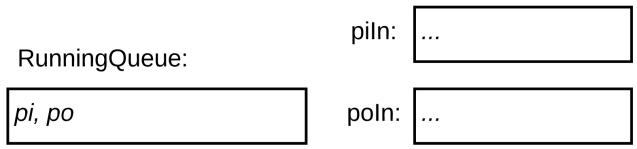
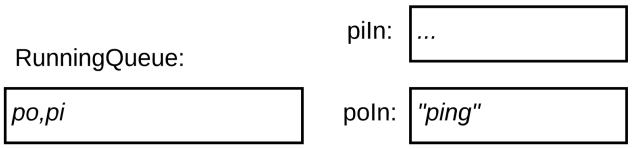
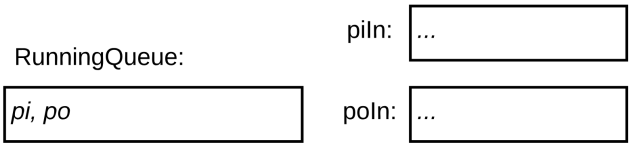
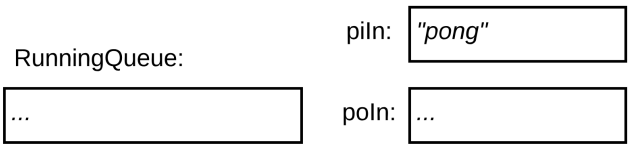


Figure 4.3: Simple Ping-Pong system with only one pair doing a single back and forth

Firstly we will detail the optimal execution trace (Table 4.1), that is the one that requires the least amount of evaluation steps before successfully running the system to completion:

Step 1. The Process System is created with an empty Running Queue and a number of `Executors` ready to consume it.	RunningQueue: ... piIn: ... poIn: ...

Step 2. Two processes are created: a `Ping` pi and a `Pong` po. Importantly the spawning process appends the processes onto the Running Queue of the Process System.	
Step 3: Now the Executors can start consuming the Running Queue. Let's say pi is picked up first and its *send* to po's InChannel is evaluated. After this its continuation (a *receive* waiting for a *pong* message) is re-appended onto the Running Queue.	
Step 4: Now say po is executed. Since *ping* has been sent to its InChannel, the *receive* operation is successful, and the continuation (a *send*) is re-enqueued onto the Running Queue.	
Step 5: Now assume that due to internal scheduling of the threads handling the Executors the continuation of po is executed. It could either be because it was re-enqueued first, or because even after being dequeued after, it reached `fastEval` faster (which is what happens in this trace, causing the Running Queue to be empty). The *pong* is sent to pi's InChannel, and then po terminates.	

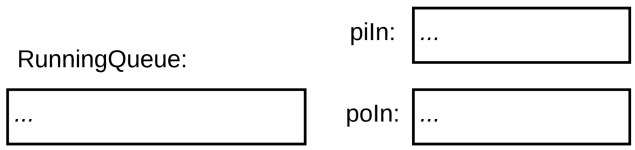
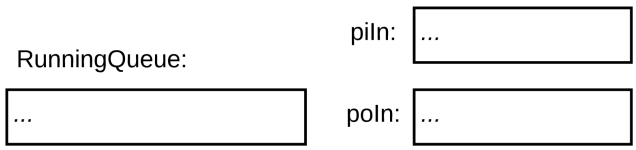
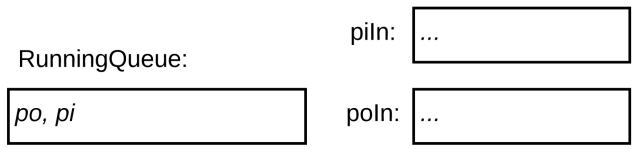
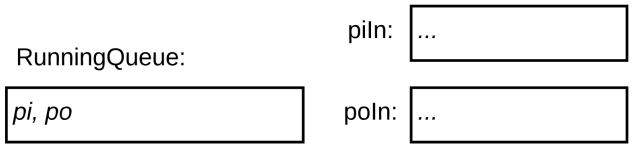
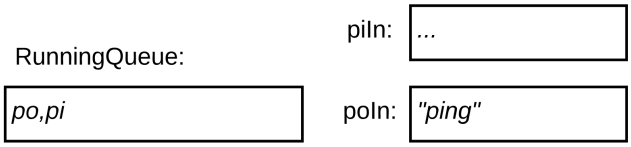
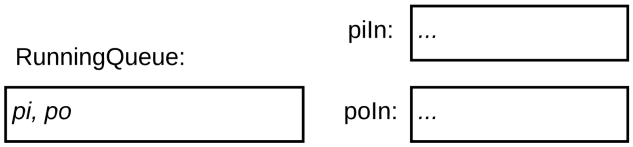
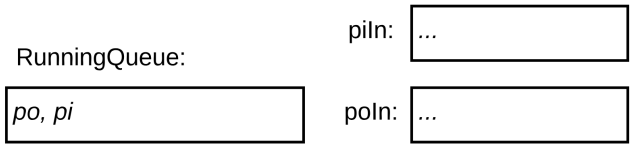
Step 6: Finally only the *receive* by pi is left, which can be executed successfully, since a message *pong* is found in its InChannel, after which pi also terminates	

Table 4.1: This is a suboptimal execution trace trace of ping-pong

This gives an idea of how the execution is run end to end to completion. However as mentioned this represents the optimal execution trace for the given Ping-Pong system. It turns out that the execution is only this efficient when for every *send-receive* pair of processes in the system at any point of the execution, the *send* is executed before. If this is not the case there will be extra steps required to reach the end, as we show below in a suboptimal execution trace (Table 4.2), where upon initialisation po is scheduled before the pi, and then again later reading *pong* is attempted before it has been sent:

Step 1. The Process System is created with an empty Running Queue and a number of `Executors` ready to consume it.	
Step 2. Two processes are created: a `Ping` pi and a `Pong` po. Importantly the spawning process appends the processes onto the Running Queue of the Process System.	

Step 3. The options are the *send* action by pi and the *receive* action by po. Assume that po is picked, the *receive* is attempted but fails because no value is found on the channel. As a result the process is re-enqueued unchanged. Notice that due to the non-deterministic nature of threads this could happen more than once.	
Step 4. The state is exactly the same as above, but now assume the *send* by pi is evaluated. This of course is successful, it sends a value to the channel shared with po, and then appends the continuation of pi, a *receive* action, back on the RunningQueue.	
Step 5. The *receive* by po is attempted again, this time successfully, which causes the value sent in the step before by pi to be received, and the continuation of po (a *send* action) to be re-enqueued.	
Step 6. The options now are a *receive* by pi and a *send* by po. Assume *receive* is once again attempted first. The channel is empty and therefore the action fails, causing the process to be re-enqueued unchanged. Once again, non-determinism could mean this happens multiple times.	

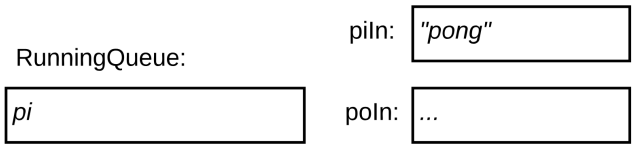
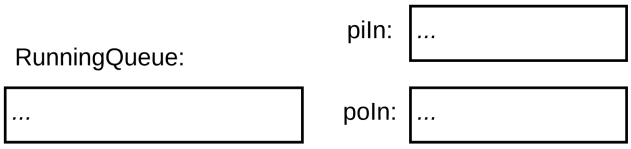
Step 7. Now assume *send* by po is executed. This sends a value to the channel shared with pi. po terminates	
Step 8. Finally the pi is executed, receiving the *pong* message successfully and then terminating.	

Table 4.2: This is a suboptimal execution trace trace of ping-pong

4.2.5 Takeaways

The key takeaway from this example is how likely failed *receive* actions can be in an execution trace. Given an execution, there is only one ideal trace, and that is the one in which there are no failed *receive* actions. This means that for each *send-receive* pair of actions, the write actions is performed first.

This is because every time a *receive* action is evaluated before a write is performed on its channel, it gets re-enqueued and will have to be re-evaluated, wasting time on the first evaluation. However since there is always the possibility of the *receive* being performed first, the likelihood of the execution being ideal decreases with the number of *send-receive* pairs.

In fact it might be even worse. This assumes that in the sub-ideal case where we unsuccessfully attempted *receive* first, we then immediately execute the *send* before retrying the *receive*. As mentioned in the example trace this is not necessarily the case. A failed *receive* could be attempted many times if for whatever reason the Executor responsible for handling the respective `write` is late or idle, and would unnecessarily slow the execution down, having Executors attempting and re-attempting the receive action and giving less opportunity to other potentially successful processes.

The inherent weakness of this rapid fire-and-miss approach is that the RunningQueue might be filled with many of this failing `In` processes with no values ready in their InChannels slowing down other processes that are ready to run. Solving this issue is what motivated the next implementation.

4.2.6 Final remarks

Looping and termination

Before moving on we are going to address the reason why dequeuing for `SchedulingQueue` (Listing 4.2) returns an option–even though the dequeuing of the Running Queue is blocking–is necessary to terminate the execution. In the original implementation, once a process evaluated to `nil` and had no future processes in its state, the thread execution would terminate. Once all threads return, the system execution is finished and the programs can terminate.

In this implementation however, the only signal that the system is terminated is that the Running Queue is left empty, as once the processes evaluate to `nil` with no future processes, nothing is rescheduled. However while this condition is necessary to establish termination, it is not sufficient to be sure that is the case. The queue could also be momentarily empty if for example not many process are left in the system, so much that they all get dequeued by an Executor, however this should not cause the system to terminate.

In order to address this situation we provided a `kill` (Listing 4.12) method to the Process System, which the user can manually call when creating a finite system. This method sets the `alive` flag to false, which is going to prevent the Executors from continuing to loop and consume the Running Queue (Listing 4.4). It also sends an interrupt to all the threads it spawned to run the Executors. Since we assume this function is called when the execution is over and the Running Queue is empty, the threads will be blocked, waiting on getting an element from the queue.

This is the reason behind the implementation of `SchedulingQueue` shown in Listing 4.2. When a sleeping thread receives the interrupt signal from the Process System it throws an `InterruptedException` which is caught and causes the queue to return `None`. This, together with the interruption of looping due to the change to `false` of the `alive` flag, in turn allows to gracefully terminate the Executors.

Listing 4.12: The method `kill` allows a process system to terminate cleaning all resources

```
override def kill(): Unit = {
  alive = false
  threads.foreach { t =>
    t.interrupt()
  }
  threads.foreach { t => t.join() }
}
```

Usage of Process System

The Akka documentation discourages from creating more than one Actor System at a time [15], but still allows it to cater for all use cases.

The same is true for Process Systems. Firstly, they are a relatively heavyweight structure, being responsible for maintaining state in memory. Secondly they hold the logic for efficient spawning of threads, however their logic doesn't account for other process systems running, because they are agnostic of each other. Creating more than one Process System could cause them to starve each other of resources.

However, while most use cases would benefit from creating only one `ProcessSystem` instance, we do allow for the creation of multiple instances to ensure any potential use case can benefit from the use of Effpi.

4.3 The WaitQueue Implementation

While the execution of input-processes does not block on receiving like the original runtime, it might still be the case that–if there are not many other processes in the Running Queue–an In process is processed multiple times while no value has been sent to its channel. Every time this happens the process is dequeued, a receive action is attempted and if it fails the process needs to be enqueued again. This could happen multiple times before the receive action is successful. This new implementation of the runtime addresses this problem and it is inspired by the way Akka's Dispatchers [17] [18] work.

4.3.1 Dispatchers in Akka

As mentioned in the introduction Akka is an actor based framework. Actors scheduling in Akka is handled by so called Dispatchers, and is based on the actors' Mailboxes [19] [20]. The key concept is that a Dispatcher stores a queue of Mailboxes, which are only scheduled when they have data available. Another consequence of this design is that Akka only pre-empts Actors execution upon input-actions.

4.3.2 Separate In process scheduling

In Effpi however we can pre-empt processes at any point, because the Executors re-enqueue a process after every step of execution. This solution allows it to provide more granularity of execution. However Akka's reactive design was an inspiration for a design that uses a double scheduling system: a general one similar to the

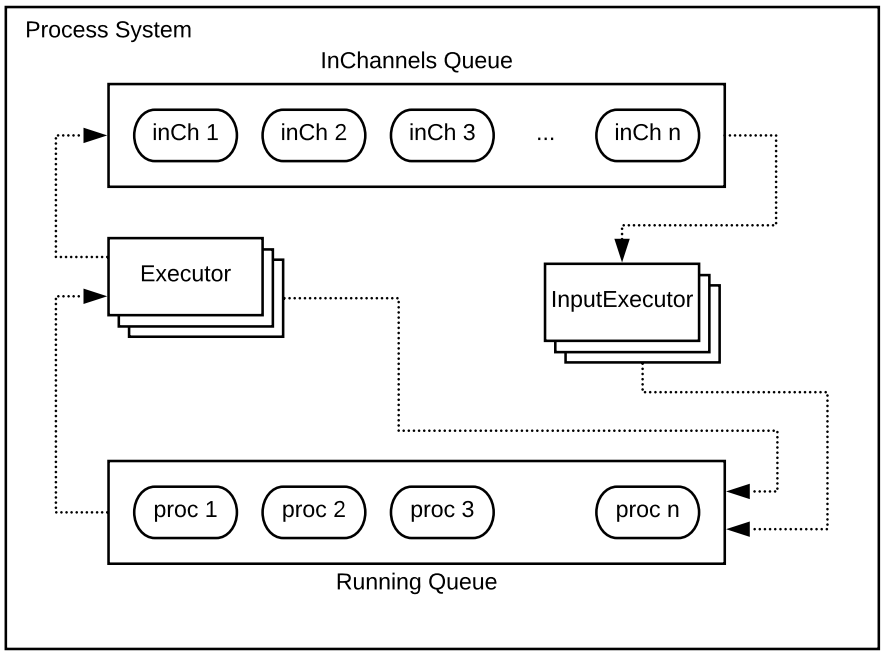


Figure 4.4: Architectural diagram of the WaitQueue implementation. The Executors consume the Running Queue and InputExecutors consume the InChannels Queue

one introduced in Section 4.2, and one specific to `In` typed processes, which would alleviate the effect of the blind rescheduling illustrated before on the execution of non `In` typed processes. We also wanted to be able to schedule `In` processes only when data is available and try to reduce the need of blind rescheduling in general.

The outline of the execution strategy illustrated in Figure 4.4 is the following:

- `InChannels` can now store a queue of In processes that want to read a value from the channel (Wait Queue).
- A new queue is added in `ProcessSystem` storing scheduled `InChannels` (InChannels Queue). A channel should be added to this queue whenever an In receiving on it can potentially be executed.
- Like in the previous RunningQueue implementation (Section 4.2), the Executors consumes the queue of processes. However now they do not execute `In` typed processes.
- `InputExecutors` consume the InChannels Queue. When a channel has a process in its Wait Queue and a value ready to be received (that is when the process can be successfully executed) it executes the the process. Then it appends the continuation back onto the Running Queue so that its execution can be continued by the Executors.

4.3.3 Changes to InChannel

Two changes are required to the `InChannel` and `OutChannel` traits and their implementation. The first one is the ability to store a queue of process states inside an InChannel. Unlike the queue in `ProcessSystem` however, this queue only really needs to store `In` type processes, which is reflected in the more stringent type.

Listing 4.13: Modifications to the `InChannel` trait for the WaitQueue implementation

```
1 trait InChannel[+A] {
2   def enqueue(i: (Map[ProcVar[_], (_) => Process], List[() => Process],
        In[InChannel[_], _, Process])): Unit
3   def dequeue(): Option[(Map[ProcVar[_], (_) => Process], List[() =>
        Process], In[InChannel[_], _, Process])]
4 }
```

Once again, internally this is implemented as a Java `LinkedTransferQueue`. The need for thread-safety comes from the fact that we want to dequeue and execute multiple waiting processes in parallel when enough data is available on the InChannel.

Listing 4.14: Implementations of `enqueue` and `dequeue` for the Wait Queue of an InChannel used by the runtime

```
1  trait QueueInChannel[+A](q: LTQueue[A]) extends InChannel[A] {
2
3    var pendingInProcesses = new LTQueue[(Map[ProcVar[_], (_) => Process],
         List[() => Process], In[InChannel[_], _, Process])]()
4
5    override def enqueue(i: (Map[ProcVar[_], (_) => Process], List[() =>
         Process], In[InChannel[_], _, Process])): Unit =
         pendingInProcesses.add(i)
6
7    override def dequeue() = pendingInProcesses.poll() match {
8      case null => None
9      case head => Some(head)
10   }
11
12 }
```

The second change needed is to be able, given a specific OutChannel to obtain its respective dual InChannel. This is reflected in the interface by adding the `dualIn` method.

Listing 4.15: `OutChannel` with the additional `dualIn` method to retireve the dual InChannel

```
trait OutChannel[-A] {
  val out: OutChannel[A] = this
  val dualIn: InChannel[Any]
  def send(v: A): Unit
}
```

Without delving into `dualIn`'s implementation, it is important to notice that each OutChannel has a single dual InChannel, and that therefore multiple calls of this method return the same object.

4.3.4 Changes to ProcessSystem

The first fundamental architectural change is the addition of a new queue of `InChannels` to `ProcessSystem` required to implement the new scheduling strategy.

The additions (highlighted in Listing 4.16) are very similar to already existing code. A queue of `InChannels`, methods to access it and additional code in `init` to initialise the `InputExecutors` that will consume the queue of `InChannels`.

Listing 4.16: Process System additions to perform the WaitQueue strategy of execution

```
trait ProcessSystem {
  var waitingInChs = new SchedulingQueue[InChannel[_]]

  def enqueueWaitingInCh = waitingInChs.enqueue
  def dequeueWaitingInCh() = waitingInChs.dequeue()

  def init(threadsPerCore: Int): Unit = {
    val numCores = Runtime.getRuntime().availableProcessors()

    val numThreads = numCores * threadsPerCore

    (1 to numThreads).foreach { c =>
      threads += new Thread(new Executor(this))
    }

    (1 to numThreads).foreach { c =>
      threads += new Thread(new InputExecutor(this))
    }

    threads.foreach{ t => t.start()}
  }
}
```

4.3.5 Changes to Executor

As a result of this new strategy, `fastEval` will need two changes in implementation, one for the `Out` case and one for `In`.

Out

`fastEval` still performs the old functionality in case of `Out`, sending a value to the OutChannel and appending the continuation of the process back onto the Running Queue. However now it also schedules the dual of the current OutChannel to allow the `InputExecutors` to consume it. The reason why this is potentially a good time to let the InputExecutors attempt to execute an `In` process and perform a *receive* is because the channel is guaranteed to have received at least one value. This argument is clearly true when just there are just one Executor and one InputExecutor, because only one `In` process can be performed at a time and there is no contention. However this still applies in a more parallel context with multiple Executors and InputExecutors with `In` processes potentially competing to consume values off a single channel. This is because for each `Out` process that is evaluated by an Executor, one value will be guaranteed to be present in the channel, and therefore if n `Out` sending to the same channel have been processed, n values will be guaranteed to be available on it. The dual is obtained using the newly added `dualIn` function illustrated in Listing 4.15, and then the channel is added to the list of waiting `InChannels` in the Process System.

Listing 4.17: `WaitQueue` implementation of `fastEval` in the `Out` case

```
1    case o: Out[_,_] =>
2      val outCh = o.channel.asInstanceOf[OutChannel[Any]]
3      outCh.send(o.v)
4      lp match {
5        case Nil =>
6          ()
7        case lh :: lt =>
8          ps.enqueueRunning((env, lt, lh()))
9      }
10     val inCh = outCh.dualIn
11     ps.enqueueWaitingInCh(inCh)
```

In

This implementation changes drastically because, as described below, `inEval` now covers the tasks that in the previous strategy were performed by `fastEval`. `fastEval`

now simply ensures that when an `In` process is found and there is a potential read action available, this is appended on the Wait Queue of the InChannel it wants to read from and that the channel itself is appended to the InChannels Queue of the Process System. This in turns means that the InputExecutors can consume the channel from the InChannels Queue, and find a process on the channel's Wait Queue. However there is still the possibility that a value has not yet been received by the channel. This is possible if a *send* action has not been performed yet.

Listing 4.18: `WaitQueue` implementation of `fastEval` in the `In` case

```
1  case i: In[_,_,_] =>
2    i.channel.enqueue((env, lp, i.asInstanceOf[In[InChannel[_], _,
       Process]]))
3    ps.enqueueWaitingInCh(i.channel)
```

4.3.6 InputExecutor

Now that the Executors contain the logic for scheduling `InChannels`, the `InputExecutors` are left with the responsibility of dealing with the actual execution of `In` processes. This means that an `InputExecutor` has a similar responsibility with respect to `In` processes to that of the `Executor` to all other process types. As the latter contains the logic to consume the Running Queue, so the former consumes the `InChannels` Queue. Consequently the logic for keeping it running forever is very similar (Listing 4.19), with a loop controlled by the `alive` flag.

However in the `InputExecutor` logic there are two queues to consider, and noticeably, the logic is somewhat inverted from the previous implementations:

- It used to be the case that `fastEval` (but also the original `eval`) would encounter an `In` process and from it then the channel from which to receive a value was established.
- In this implementation, on the other hand, the channel is established first when it is consumed by the InputExecutor, and only then a process that wants to perform a read over it is dequeued from the queue contained in the channel itself.

The first queue encountered is the Process System's InChannels Queue. Dequeuing has a similar behaviour to Running Queue. In practice dequeuing is blocking and therefore a value is always retrieved. The only exception is on termination for finite systems, for similar reasons to those detailed in Section 4.2.6.

If a channel is returned, then the logic can continue leading to the second queue, the Wait Queue of the channel. If the Wait Queue is empty and a process cannot be

dequeued then the evaluation simply continues onto the next available InChannel. However, if a process is found, then `inEval` is called to perform the evaluation.

Listing 4.19: `WaitQueue` logic to consume the InChannels Queue

```
override def run() = {
  while (ps.alive) {
    val maybeInCh = ps.dequeueWaitingInCh()
    maybeInCh match {
      case Some(in) =>
        in.dequeue() match {
          case Some(proc) =>
            inEval(proc)
          case None =>
            ()
        }
      case None =>
        ()
    }
  }
}
```

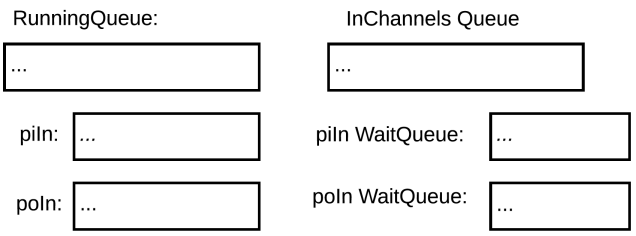
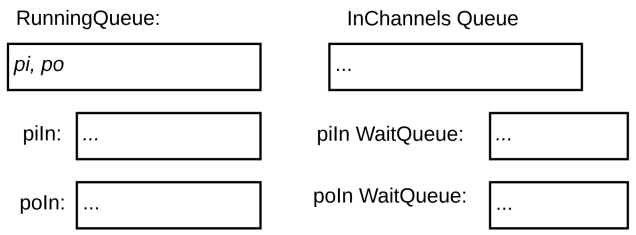
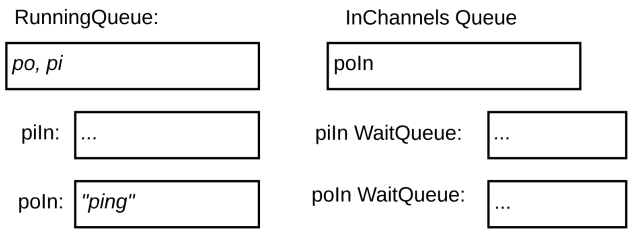
The implementation of `inEval` (Listing 4.20) is very similar to that of the `In` case in `fastEval` for the RunningQueue implementation. If a value is successfully polled from the channel, then the continuation of this process is enqueued back onto the Running Queue of the Process System. In case the channel has not received a value however the behaviour is now different. The process is re-enqueued onto the Wait Queue of the InChannel, so that it can be retried in the future and the channel itself is rescheduled.

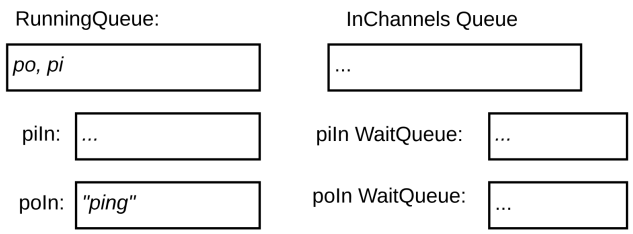
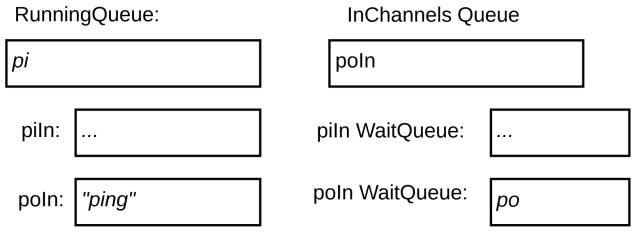
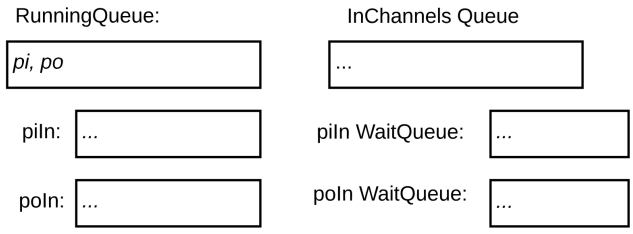
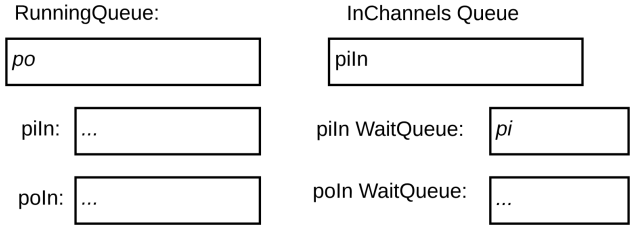
Listing 4.20: `WaitQueue` implementation of `inEval` to execute `In` processes

```
def inEval(proc: (Map[ProcVar[_], (_) => Process], List[() => Process],
    In[InChannel[_], _, Process])): Unit = {
  val (env, lp, i) = proc
  i.channel.poll() match {
    case Some(v) =>
      val cont = i.cont.asInstanceOf[Any => Process](v)
      ps.enqueueRunning((env, lp, cont))
    case None =>
      i.channel.enqueue((env, lp, i))
      ps.enqueueWaitingInCh(i.channel)
  }
}
```

4.3.7 WaitQueue strategy on Ping-Pong example

We use an example execution trace of the Ping-Pong system introduced in Listing 3.1 and used in Section 4.2.4 to illustrate how this strategy works (Table 4.3). Once again, we do so without explicit assumptions on the hardware, just that it can perform parallel execution. It is useful to focus on how this implementation deals with failed *receive* actions compared to the RunningQueue one.

Step 1. The Process System is created with an empty Running Queue and a number of `Executors` and `InputExecutors` ready to consume it.	
Step 2. The two processes pi and po are spawned and added to the Running Queue.	
Step 3. The two available options now are a *send* for pi and a *receive* for po. If *send* is executed, a *"ping"* message is sent to po's InChannel, which is then appended onto the Process System's InChannels Queue. Then pi's continuation is rescheduled.	

Step 4. In this implementation the state is more complex: the InChannels Queue is also being consumed and as such it requires consideration when analysing the execution. There is now the *receives* from pi and po on the Running Queue and po's channel on the InChannels Queue. Assume the `InputExecutor` is faster, and the channel is consumed first. The *receive* action will be unsuccessfully attempted after the channel's Wait Queue is found to be empty. In this case the channel is not rescheduled and the execution continues.	
Step 5. Now the only available choices are the *receives* from pi and po. the latter is executed first, and po's channel is scheduled, with po appended to its Wait Queue.	
Step 6. The options now are *receive* from pi on the Running Queue and the channel from po on the InChannels Queue. The latter is executed first, and since a value was sent to the channel in step 3 and the *receive* process is still to be processed the execution is successful. The continuation of po, a *send* is rescheduled onto the Running Queue.	
Step 7. Now there are a *receive* from pi and a *send* from po. This time *receive* is executed first. this causes pi to be appended on the Wait Queue of its InChannel, and the channel itself to be appended to the InChannels Queue.	

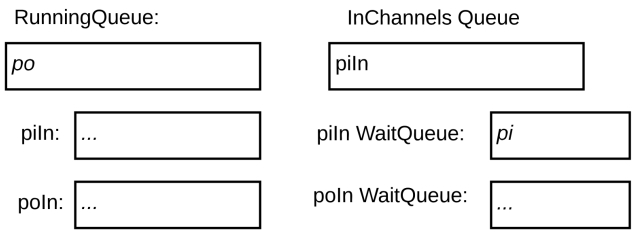
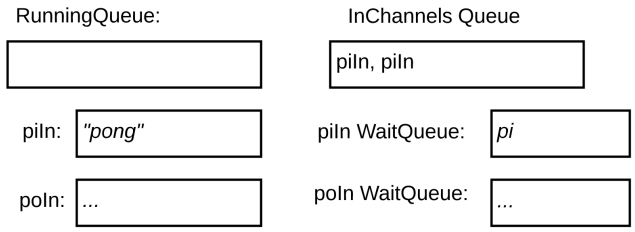
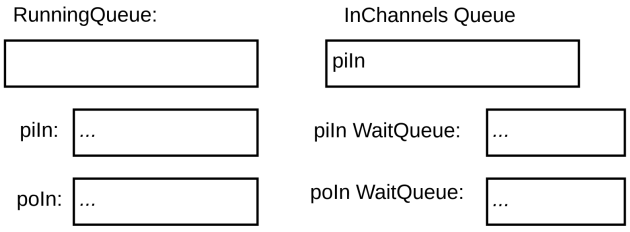
Step 8. The options are now *send* from po or pi's InChannel. Assuming the InputExecutor is faster the latter is considered first. Since no value has yet been sent to the channel the execution is unsuccessful. However this time the channel is re-scheduled to the InChannels Queue. Note that this unsuccessful case, unlike that in step 4, can be repeated an arbitrary number of time depending on the internal scheduling of the threads running the Executors and the `InputExecutors`.	
Step 9. The *send* from po is now executed, it sends a value to pi's InChannel, and the channel itself is scheduled. At this point po is finished.	
Step 10. Notice that now the InChannel for pi is present twice on the InChannels Queue. However this is irrelevant from a correctness perspective. Both are picked up by InputExecutors, but only the first one to retrieve the single process from the queue will be successful. The other one will simply flush out the other copy, as the *receive* fails and the channel will not be rescheduled (the state is not shown it it would leave all the queues empty).	

Table 4.3: This is a suboptimal execution trace trace of ping-pong

4.3.8 Takeaways

While this strategy was originally supposed to remove the blind rescheduling of *receives*, it might still schedule redundant empty `InChannels`. Given a *send-receive* pair,

if the `Out` process is executed first, then indeed the channel is not rescheduled and the execution just waits for a dual `In` before rescheduling the channel. However if the `In` is executed first, then the InputExecutors will keep rescheduling the channel, every time they dequeue one that has no value available to be read.

Incidentally from the trace this seems redundant, as once the `Out` process is executed it also schedules a second reference of the InChannel onto the InChannels Queue. This ends up leaving an extra copy of the channel on the queue, which potentially might never be removed as more references of each channel will added than are removed. This might eventually cause memory issues at scale.

4.3.9 Early WaitQueue implementation leading to deadlock

Unfortunately however, while the rescheduling of the channel inside the InputExecutor does seem at a first glance not only redundant but a clear hindrance to performance, it is necessary for correctness. Without it there is the possibility for a subtle deadlock scenario which unfortunately precludes completely the possibility of using this strategy without any sort of blind rescheduling.

Consider the following example:

1. A *send-receive* pair is given, and the `In` process in is picked up first by an Executor, its channel c scheduled and then picked up by an InputExecutor.
2. A InputExecutor finds c, dequeues in and then attempts to read a value from c. This is unsuccessful, however assume that by virtue of the host hardware scheduling of the threads, the context switches before it can re-append in onto c's Wait Queue.
3. Now an Executor picks up the respective `Out` process out causing a value to be sent to c before it is again scheduled onto the InChannels Queue.
4. A second InputExecutor takes c from the InChannels Queue and attempts to dequeue an `In` process from it. This of course is unsuccessful as the first InputExecutor has not re-enqueued it yet. At this point this InputExecutor simply continues consuming the InChannels Queue forgetting c
5. Now back to the first InputExecutor, in is re-appended onto c's Wait Queue and then, without rescheduling, the InputExecutor continues its execution forgetting c

At this point in this scenario a value is present for c to read, and its Wait Queue is not empty, meaning there are `In` processes that want to read that value, however the channel has been forgotten and will not be scheduled again. Rescheduling upon failure to find a process in c's Wait Queue solves the issue. In conclusion this implementation mitigates, but does not completely solve the failed `receives` issue.

4.3.10 Final Remarks

One last thing worth noting is that given an InChannel c the `In` processes trying to read from it are gradually appended to its Wait Queue in order by `fastEval`. However, in the case that no `Out` process has yet been processed for these `Ins`, when the initial attempt of `inEval` triggered by the addition of c by the `In` case of `fastEval` fails due to no value found on the channel, the processes will be re-appended, but the concurrent nature of the scheduling of the threads running the InputExecutors means that the original order is not guaranteed to be maintained.

While this has some effect in terms of fairness and performance, it does not affect the correctness of the runtime. The fact that the Wait Queue is checked before trying to get a value from the channel is intentional and important. Doing the opposite – trying to read a value from the channel before checking for processes trying to read it – would make correctness much harder to achieve.

The fact is, these processes are inherently concurrent and therefore in what order they are evaluated does not actually matter. Yet the same is not true for values sent to a channel. If two parallel processes p and q both send a value to channel in, the order in which they are received is irrelevant. However in the case in which q is the continuation of p, the order is very important.

This is why this strategy avoids reinsertions of values on the queue of values an InChannel, and elects to do so on the Wait Queue, because the latter is more flexible, making it more appropriate to use for checks.

4.4 Improved WaitQueue implementation

While the current implementation of `InputExecutor` reschedules the `InChannels` when failing to poll a value from it, another possibility is to do so when failing to find an `In` process in the channel's Wait Queue (the change from the previous implementation is highlighted in Listing 4.21).

Listing 4.21: `WaitQueueImproved` implementation of `InputExecutor`

```
1  override def run() = {
2    while (ps.alive) {
3      val maybeInCh = ps.dequeueWaitingInCh()
4      maybeInCh match {
5        case Some(in) =>
6          in.dequeue() match {
7            case Some(proc) =>
8              inEval(proc)
9            case None =>
10             ps.enqueueWaitingInCh(in)
```

```
        }
      case None =>
        ()
    }
  }
}

def inEval(proc: (Map[ProcVar[_], (_) => Process], List[() => Process],
    In[InChannel[_], _, Process])): Unit = {
  val (env, lp, i) = proc
  i.channel.poll() match {
    case Some(v) =>
      val cont = i.cont.asInstanceOf[Any => Process](v)
      ps.enqueueRunning((env, lp, cont))
    case None =>
      i.channel.enqueue((env, lp, i))
  }
}
```

It is also possible in this strategy to omit scheduling the channel in the `In` case of `fastEval`

Listing 4.22: `WaitQueueImproved` implementation of `fastEval` in the `In` case

```
case i: In[_,_,_] =>
  i.channel.enqueue((env, lp, i.asInstanceOf[In[InChannel[_], _,
      Process]]))
```

This change does not actually eliminate the failed *receives* issue, it simply changes the scenario in which they occur. Previously this was when an `In` was scheduled before any `Out` on its channel. Now this happens when an `Out` is scheduled before any `In`.

However one potential advantage that comes from not scheduling the channel in `fastEval` in the `In` case is that channels are never double-scheduled, as it was happening before. The exact dynamic why this is the case may not be intuitive, so we produce two short traces (one with the old strategy and one with this new one) to illustrate the difference. Assume two processes out and in with the first one sending a message to the second one, which reads the value through its InChannel c.

With the original WaitQueue implementation:

1. in is processed by an Executor. in is appended onto c's Wait Queue and c is scheduled onto the Process System's InChannels Queue.

2. out is processed now. A value is send to in and c is scheduled again.

3. A InputExecutor picks up c and successfully executes in since a value is available to be read and there is in ready to receive it.

4. The leftover copy of c remaining on the InChannels Queue will need to be flushed out by a InputExecutor. If this were in the context of a longer execution, this would add an extra step to it.

With this current improved version:

1. in is processed by an Executor. in is appended onto c's Wait Queue.

2. out is processed now. A value is send to in and c is scheduled onto the Process System's InChannels Queue.

3. A InputExecutor picks up c and successfully executes in since a value is available to be read and there is in ready to receive it.

This implementation takes one less step to fully execute. This is shown in Section 5.3 to affect performance at scale as it makes more efficient use of resources.

4.5 Multi-step Implementation

4.5.1 Multiple steps evaluation

This implementation takes a different directions and addresses the cost of *context switching* between two consecutive calls to `fastEval`. As mentioned before `fastEval` only performs one step of the execution of each process at a time. In between calls it dequeues and enqueues elements on the Running Queue. This abundance in bookkeeping operations and frequent access to this queue which is stored in memory and shared by multiple threads definitely represents an inefficiency in the execution.

Obviously some of this bookkeeping operations are required, because it is fundamental for the correctness of the runtime that the general state of the system is maintained. Also some degree of context switching is necessary to provide fairness and allow all processes to gradually execute.

However one possible solution would be to reduce the frequency of context switching to reduce its relative impact in performance. Reducing the time spent performing context switching, which per se does not lead to advance in the execution, would reduce the overall running time.

4.5.2 Multi-step Executors

Our solution to do this was to allow the Executor to make several calls to `fastEval` once a process is dequeued from the Running Queue, in order to perform multiple steps of the execution. In practice this meant making `fastEval` tail recursive, adding an extra argument `stepsLeft` to keep track of how many iterations are left (Listing 4.23). Once this counter gets to zero, the base case of the function is reached and the state of the continuation is rescheduled on the Running Queue.

This change was made to all the implementations described in earlier sections (Section 4.2, Section 4.3 and Section 4.4), however we will take the Improved WaitQueue implementation as example to illustrate the changes.

Listing 4.23: `fastEval` from the WaitQueueImproved implementation with multi-step execution

```
@annotation.tailrec
private def fastEval(
  proc: (Map[ProcVar[_], (_) => Process], List[() => Process], Process),
  stepsLeft: Int
): Unit = {
  if (stepsLeft <= 0) {
    ps.enqueueRunning(proc)
  } else {
    val (env, lp, p) = proc
    p match {
      case d: Dep[_,_] => ...
      case i: In[_,_,_] => ..
      case o: Out[_,_] => ...
      case f: Fork[_] => ...
      case n: PNil => ...
      case y: Yield[_] => ...
      case d: Def[_,_,_,_] => ...
      case c: Call[_,_] => ...
      case s: >>:[_,_] => ...
    }
  }
}
```

The average change to each case is fairly minimal, mostly just substituting a previous call to `ps.enqueueRunning(contState)` with `fastEval(contState, stepsLeft-1)`, where `contState` represents the state of the continuation process. This is the case for example in the `Out` case:

Listing 4.24: `Out` case of WaitQueueImproved `fastEval` with multi-step execution

```
case o: Out[_,_] =>
  val outCh = o.channel.asInstanceOf[OutChannel[Any]]
  outCh.send(o.v)
  val inCh = outCh.dualIn
  ps.enqueueWaitingInCh(inCh)
  lp match {
    case Nil =>
      ()
    case lh :: lt =>
      fastEval((env, lt, lh()), stepsLeft - 1)
  }
```

However the number of iterations left is not strict. In fact, in case of `In` the behaviour remains the same as the modified InputExecutors implementation, and no further recursive call is made. This is because the responsibility for handling `In` processes and all other types is still separated to InputExecutors and Executors respectively, and therefore once the Executor is forced to yield the process because it needs to perform a *read*, it should just continue consuming the Running Queue.

Listing 4.25: `In` case of WaitQueueImproved `fastEval` with multi-step execution

```
case i: In[_,_,_] =>
  i.channel.enqueue((env, lp, i.asInstanceOf[In[InChannel[_], _,
      Process]]))
```

4.5.3 Multi-step InputExecutors

Similarly the same can be done to the InputExecutors, to make the evaluation of `In` processes multi-step. The caveat is that a new iteration is attempted only in case of successful `receives`. This means that as soon as an attempt fails, whether because there are no processes in the channel's Wait Queue, or because there is no value to read on the channel, the recursive execution stops and the context switches.

Listing 4.26: `inEval` from the WaitQueueImproved implementation with multi-step execution

```
@annotation.tailrec
private def inEval(
  proc: (Map[ProcVar[_], (_) => Process], List[() => Process],
      In[InChannel[_], _, Process]),
  stepsLeft: Int
): Unit = {
  if (stepsLeft == 0) {
    ps.enqueueRunning(proc)
```

```
8     } else {
9       val (env, lp, i) = proc
10      i.channel.poll() match {
11        case Some(v) =>
12          val cont = i.cont.asInstanceOf[Any => Process](v)
13          cont match {
14            case i: In[InChannel[_], _, Process] =>
15             inEval((env, lp, i), stepsLeft - 1)
16
17            case _ =>
18              ps.enqueueRunning((env, lp, cont))
19          }
20        case None =>
21          i.channel.enqueue((env, lp, i))
22      }
23    }
```

While this implementation is likely to bring improvements in performance, it is also important to notice that allowing for too many consecutive iterations reduces the granularity in the concurrency of the execution. Granularity is an important property to have in a framework for concurrent programming aiming not to be domain specific, because some application need it to resemble real life systems. A default value of 10 consecutive iteration steps is established, however we provide the possibility to tweak this value, through a parameter in the `ProcessSystem` constructor.

4.6 Abandoned design: SleepingMap

Prior to any of the WaitQueue implementations, we explored another strategy to deal with failed *receives*, illustrated in Figure 4.5. Instead of storing a queue of process states into each *InChannel* and having a global (as in accessible to all threads) queue of `InChannels` to be consumed by a scheduling agent, this designed kept into the `ProcessSystem` a map from `InChannel` to list of *sleeping processes* states (Sleeping Map). An `In` process that performed an unsuccessful *receive* was considered ready to be put to sleep, and enqueued to the list of processes of the channel it tried to read. A *sleeping process* would only be awaken after a *send* to the channel it needs to read from had been performed, providing a chance of successful execution.

4.6.1 Map of sleeping processes

Firstly, in order to implement this strategy we need a map that can store a list of `In` process states grouped by their InChannel. It also needs to be thread safe, since it will be accessed concurrently by the Executors.

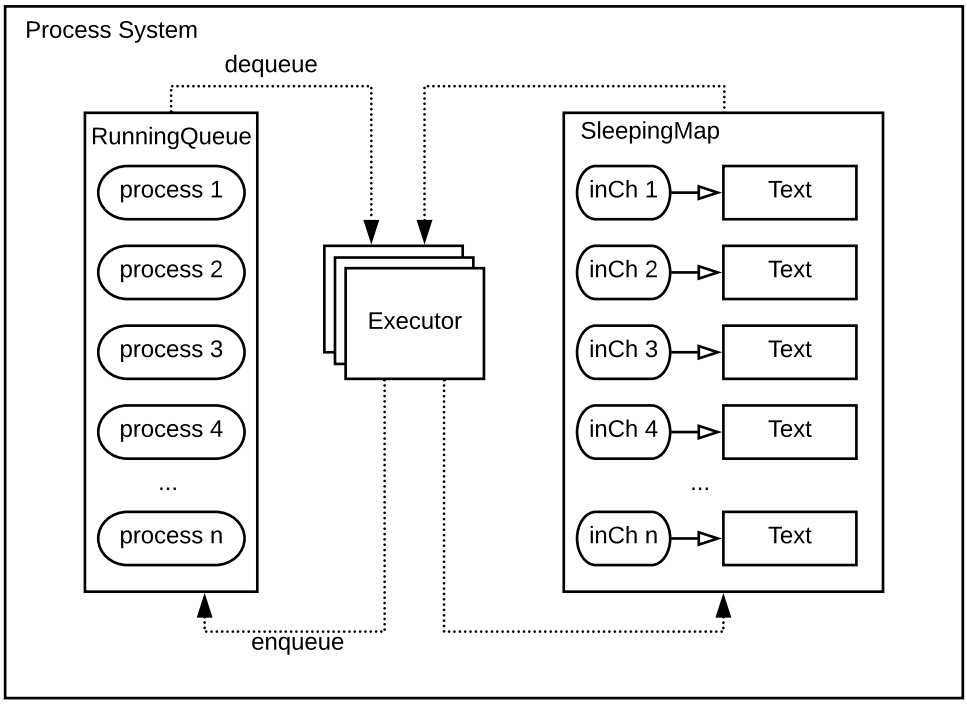


Figure 4.5: Architectural diagram of the SleepingMap implementation. The executors here consume and reschedule processes from a queue in memory

The main methods are `put` and `recover`. The former allows us to either append a process onto a channels list or, if no prior mapping for that channel was found, to create a fresh one with a list containing the new *sleeping process*.

The latter deals with awaking processes, and due to the maps structure has a slightly more complex behaviour. A mapping is only returned if at least one sleeping process is found. When a process state is recovered it is also removed from the list, and if the list is empty the mapping is removed completely from the map. This function has multiple steps and checks, but it is important for the overall correctness that this whole function is executed atomically. Unfortunately the only way to guarantee this is to wrap the whole function body in a `synchronized` block. This may have a negative effect on performance.

Listing 4.27: `SleepingMap` used to store input-processes waiting on a value to read

```
1 class SleepingMap[K, V] {
2
3   val map = (new ConcurrentHashMap[K, List[V]]).asScala
4
5   def put(key: K, value: V): Unit = this.synchronized {
6     map.putIfAbsent(key, List(value)).foreach { v =>
7       map.replace(key, v :+ value)
```

```
8       }
9     }
10
11    def recover(key: K): Option[V] = {
12      this.synchronized {
13        map.get(key) match {
14          case Some(v) =>
15            v match {
16              case Nil =>
17                map.remove(key)
18                None
19              case x :: Nil =>
20                map.remove(key)
21                Some(x)
22              case x :: _ =>
23                map.replace(key, v.tail )
24                Some(x)
25
26            }
27          case None =>
28            None
29        }
30      }
31    }
32
33  }
```

4.6.2 Changes to the Executor to deal with sleeping processes

In order to actuate this scheduling strategy, changes need to be made to the `In` and `Out` cases of `fastEval`. Since all other types are not cannot be sleeping (like `In`), nor can they cause a *sleeping process* to wake up, they remain unchanged. It is worth remembering that this implementation was intended as an improvement on the RunningQueue design, and so by 'unchanged' we mean that those cases maintain the same implementation of that strategy.

In

If a value is successfully retrieved from the channel, then the implementation is unchanged, in that it reschedules the state of the continuation. However if the channel could not provide any value, then the process is put to sleep on the map, and it is not rescheduled.

Listing 4.28: `In` case of `fastEval` using the Sleeping Map to handle input-processes

```
case i: In[_,_,_] =>
  i.channel.poll() match {
    case Some(v) =>
      val cont = i.cont.asInstanceOf[Any => Process](v)
      ps.enqueueRunning((env, lp, cont))
    case None =>
      val inCh = i.channel.asInstanceOf[InChannel[Any]]
      val inProc = i.asInstanceOf[In[InChannel[_], _, Process]]
      ps.putToSleep(inCh, (env, lp, inProc))
  }
```

Out

In case of `Out` first a value is sent and the continuation of the current process is rescheduled, just like in the RunningQueue implementation. Then however the dual channel of the OutChannel to which the value has been sent is obtained (using the `dualIn` function described in Section 4.3) we attempt to awake a sleeping process waiting on a value from that channel. Whether a process will be found depends on the scheduling order, and if any `In` processes had been previously attempted to be executed. If such a process is found, it is immediately executed.

Listing 4.29: `Out` case of `fastEval` using the Sleeping Map to handle input-processes

```
case o: Out[_,_] =>
  val outCh = o.channel.asInstanceOf[OutChannel[Any]]
  outCh.send(o.v)
  lp match {
    case Nil =>
      ()
    case lh :: lt =>
      ps.enqueueRunning((env, lt, lh()))
  }
  val inCh = outCh.dualIn
  val inProc = ps.awakeSleeping(inCh)
  inProc match {
    case Some(proc) =>
      fastEval(proc)
    case None =>
      ()
  }
```

The reason to do this instead of, for example, rescheduling the `In` process on the Running Queue, is that it decreases the likelihood that a fresh `In` channel steals the value from the channel first causing the newly awaken process to go back to sleep.

While this could potentially mean that it becomes more likely for all `In` to be briefly put to sleep, it should decrease the likelihood of starving a single `In` process and keeping it asleep for a long time.

4.6.3 How this leads to deadlock in the Ping-Pong example

While this strategy may work in some cases, it is not inherently safe. Below is an example trace of such a scenario with two Executors ($s1$ and $s2$) handling the execution of Ping pi and Pong po:

1. $s1$ picks po from the Running Queue. It attempts to read a *ping* message, but none was sent. Now imagine that due to the inherent scheduling of threads of the host hardware the context switches before po can be put to sleep.
2. $s2$ picks up pi and sends a *ping* message to po's InChannel. It then checks if for this InChannel there are any sleeping processes, but none are found and therefore scheduled. The continuation of pi, a *receive* action for a *pong* message is scheduled.
3. The context goes back to $s1$, which now puts po to sleep and which will now wait to be awaken.
4. The system is now hung. po has a value available but it has not been rescheduled, so it will never be able to read it. In turns pi is waiting for po to send a *pong* message, but this is conditional on po receiving a *ping* first. Therefore the system is in deadlock.

While this does not necessarily precludes the use of this general design, it unfortunately makes this implementation unusable, and caused us to abandon the design.

Chapter 5

Evaluation

In this chapter we first review the changes that surfaced to the level of the public API, and what effect this has on the library, especially in terms of the correctness guarantees it can provide. We also review and justify the design choices made throughout the implementation that lead to the final design. We then analyse the performance of the optimised implementations of the runtime introduced in Section 4 and compare it to that of the original runtime. This allows us to highlight the improvements made both in terms of scalability and performance across the board. We also compare the different optimised implementations to identify their relative strengths and weaknesses and provides guidance to choose the best fit for a given use case. Finally we provide the results of fine-tuning the default value of the *thread-per-core* parameter, to justify our choice heuristically.

5.1 Changes to the public API

A fundamental prerequisite of this project was to provide an optimised implementation of Effpi's runtime which could improve on the original implementation's performance, without trading off the correctness guarantees which the library can provide. In Section 1.2 we established that the way to do so was to minimise changes to the DSLs and the types they provide, as those are the part part of the library grounded on the theory that allows to provide those guarantees.

Fortunately the runtime implementation proved to be an orthogonal part of the architecture, and the typed structures created through the DSLs could be executed without requiring any modification. This means that the optimised runtimes presented in Section 4 are able to provide the same guarantees as the original implementation.

There were a couple of changes, independent of the DSLs, that surfaced to the public API level:

- One is the necessity to explicitly instantiate a Process System before being able to start spawning processes (or actors).
- The other one is the need to explicitly call the `kill` method of the Process System to terminate its execution.

While we considered to provide a default implicit Actor System that could be available through imports (similarly to how one can use the default ExecutionContext when using Futures and Promises in Scala), we decided that this was not necessarily a bad change, given that Akka also requires the explicit instantiation of Actor Systems [15].

5.2 Attempted and abandoned design choices:

Throughout the project there were two designs initially adopted that had to be later abandoned. In both cases this was due to a lack of correctness in the implementation that would lead to deadlocks under certain conditions. This of course is not acceptable as any design should provide some improvement in performance, but has the definite prerequisite of correctness.

In the case of the implementation that makes use of a sleeping process map (Section 4.6), the issue was due to not enough atomicity in the implementation of fastEval. The design itself is not necessarily completely flawed, however to ensure atomicity much of the code would have to be executed in `synchronized` blocks, which would likely be detrimental to performance. The advantage of the two WaitQueue strategies is that they avoided the issue in a slightly more elegant and flexible way, with each channel being responsible of its own *sleeping processes* as opposed to a global data structure. Furthermore it is a more tried approach, as it was inspired by Akka's Mailboxes and Dispatchers, an industry standard design to handle this type of problem.

The other faulty design that was initially attempted was InputExecutors without blind rescheduling within `inEval`. This solution had the advantage of only scheduling when `fastEval` encountered an `In` or an `Out` process. Unfortunatelly, as illustrated in Section 4.3.9, this implementation allowed the runtime to cause deadlocks under specific circumstances, and the only readily available fix unfortunately reintroduced the problem of failed *receives*. While as we will see this is generally still an improvement on the RunningQueue implementation, and it is further improved by the WaitQueueImproved implementation, it misses to solve a significant bottleneck to the runtime performance. Possible further improvements that would better address this issue are presented in the future work section.

5.3 Performance

First of all we analyse the performance of the different optimised implementations (referred to as **RunningQueue**, **WaitQueue** and **WaitQueueImproved**) and compare them to the original runtime (referred to as **Original**). The analysis is done using a number of different benchmarks, each testing a specific facet of the runtime. All of the benchmarks utilised for this analysis are based on the Savina Benchmark Suite [13], which is widely used to test concurrency frameworks. **RunningQueue**, **WaitQueue** and **WaitQueueImproved** all benefit from some level of multi-step execution, more specifically both the Executors and the InputExecutors have their number of maximum consecutive steps of iteration set to 10. Also note that all these benchmarks have been run on a work station with the following characteristics:

- hardware specifics: 4 x Intel Core i7-4790 CPUs (3.60GHz), 8 hyperthreads, 16 GB RAM
- software installed: Java 1.8.0_172 (HotSpot 64-Bit Server VM), Dotty 0.7.0-RC1, Ubuntu 16.04.

and that the results shown are the average of repeating each benchmark 30 times to account for variance.

5.3.1 Chameneos

Overview

Chameneos [13] [14] models a peer-to-peer system. It features a central broker with a public address (or channel in our case) which handles connecting the peers to each other. Then there are the Chameneos, that is the peers of the system.

Each chameneos first communicates to the broker its wish to be paired with another chameneos. The broker waits for two chameneos to make such a request, and then provides each with the other's address. At this point the two chameneos will send each other their respective colour and start over. Once the chameneos receive the other's colour they use that information to mutate their own colour for the next meeting (the details of how the colour mutation algorithm works are actually irrelevant for the purpose of this benchmark, as long as all the information it needs are the colours of the two mated chameneos). A meeting is defined to be each time the broker pairs two chameneos. The benchmark terminates after a specific number of meetings is reached. Note that for this specific benchmark the number of meetings allowed is 25000 which is half the maximum number of chameneos tested. Since each meeting involved two chameneos, this number gives at least a theoretical

chance to all chameneos to meet at least once, even with the largest system tested, which involves 50000 chameneos.

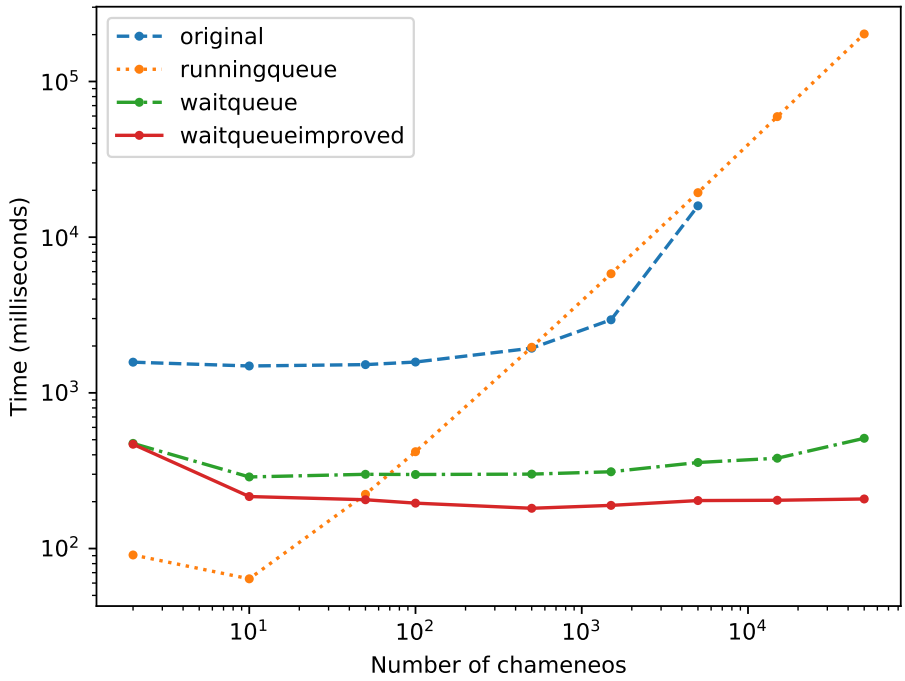


Figure 5.1: Chameneos system: time of execution against number of chameneos. Both scales are logarithmic. 25000 meetings allowed.

Results analysis

Figure 5.1 shows the results for the different implementations. The first thing worth noting is how **Original** crashes with systems larger than 5000 chameneos circa. We will see that this is about half the number for which it crashes in other benchmarks. This is because under the hood the implementation spawns two concurrent processes for each chameneos. This was necessary for two reasons:

- Since these benchmarks are originally meant to test actor based frameworks, they have been implemented using the ActorDSL. Actors are supposed to have a single Mailbox, and a Mailbox (which under the hood is an `InChannel`) can only accept a specific type of message. However each chameneos needs to deal with a message from the broker establishing the p2p connection, and one from another chameneos sharing its colour.
- The implementation needed to keep into account that one of the two chameneos could be executed faster than the other. Given a pair of chameneos $c1$ and $c2$, $c1$ might receive $c2$'s address and send it its colour before $c2$ has even received a response from the broker. Therefore the ability to receive the mate's

colour should not be dependent on having received the mate's address from the broker.

The full code of Chameneos referred to in this description can be found in Appendix A.

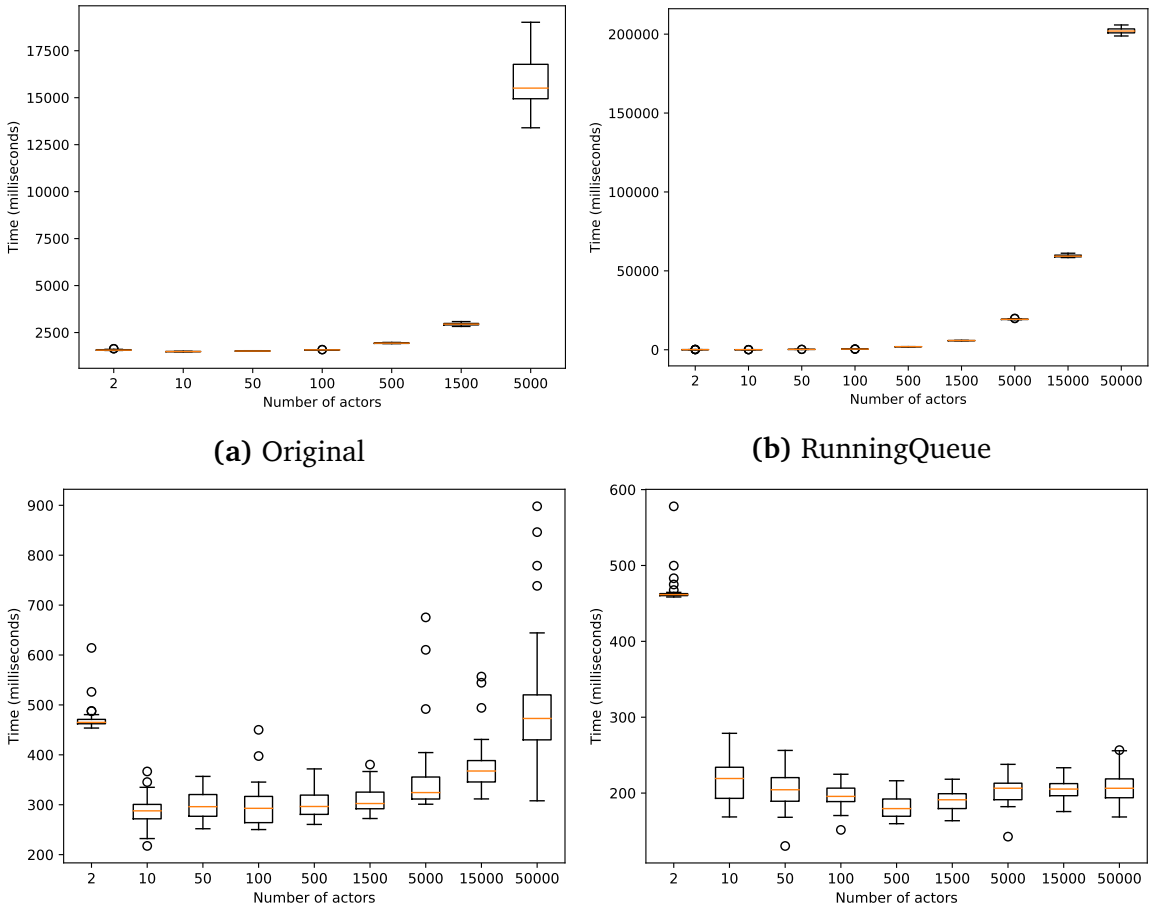
Another interesting fact about **Original** is that, at least initially, it performs significantly worse than all other implementations. This is likely due to the large cost for context switching with JVM threads.

Regarding **RunningQueue**, it steeply slows down as the number of chameneos increases, to the point that **Original** catches up to it before crashing. This is likely due to the fact that failed *receives* are appended back into an increasingly large queue which therefore causes an explosion in complexity combined with the bottleneck caused by the broker. If the broker process fails a *receive*, the whole system needs to wait for it to make it to the front of the Running Queue before execution can continue.

Lastly, while both **WaitQueue** and **WaitQueueImproved** show a reasonable performance, **WaitQueueImproved** seems to perform consistently better by what seems to be a constant factor. This could be due to the double-scheduling that occurs in **WaitQueue**. While these copies are eventually flushed out, they may cause the InChannels Queue to be slightly larger at any point in time. This has two effects:

- Having these unnecessary copies of InChannels that need to be flushed out reduces the relative time InputExecutors spend executing useful channels, thus increasing the overall execution time.
- The queue is internally implemented using Java's LinkedTransferQueues, which under the hood implements all enqueue/dequeue operations using the same private method `xfer` [7]. This method performs internally performs traversal of the queue (although it is optimised to a certain extent). This means that the performance of appending to a LinkedTransferQueue, and as a result to the InChannels Queue and to the Running Queue, will be affected by the queue size. The fact that **WaitQueue** keeps the queue size constantly larger than **WaitQueueImproved** will cause scheduling of process states or channel to be slightly less efficient.

We look now at the individual results to asses their variance (Figure 5.2). It seems that the original runtime only varies significantly at the end of the plot just before a crash occurs. This indicates that this implementation becomes less reliable with larger systems. Given that for lower values it was outperformed by the optimised implementation, **Original** simply does not seem like a good solution to handle p2p systems.



(a) Original

(b) RunningQueue

(c) WaitQueue

(d) WaitQueueImproved

Figure 5.2: Performance results for the Chameneos benchmark

It is hard to accurately asses the variance of **RunningQueue**, because its performance degenerates so dramatically that in comparison the deviation results fairly small. However in this case variance is of little relevance, as the performance is so considerably worse that the **WaitQueue** and **WaitQueueImproved** implementations (by almost a factor of $\times 1000$) that it is clear that this implementation is not an optimal solution to handle p2p systems.

Looking at Figure 5.1 we established that **WaitQueueImproved** seems to perform slightly, but decisively, better than **WaitQueue**. However their performance is similar enough that variance could make a difference. Large variance is not positive, as it means unpredictability. If a solution performs better on average, but has extreme variance in its results, it might actually be less convenient to adopt if worse case performance is a concern. However in this case we can see that on average, but especially with larger systems, **WaitQueueImproved** seems to have lower variance in results, which establishes it as the likely best implementation for p2p systems.

5.3.2 Counting Actor

Overview

Counting Actor [13] [1], unlike the other benchmarks, only spawns two processes, one of which sends N consecutive messages to the other. Its goal is to test throughput as the number of messages changes.

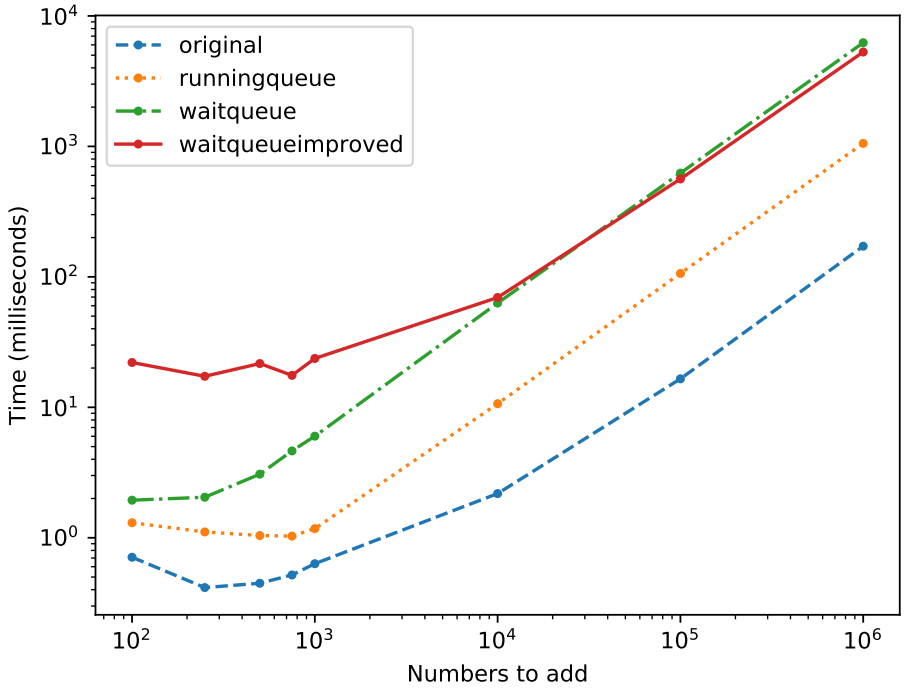
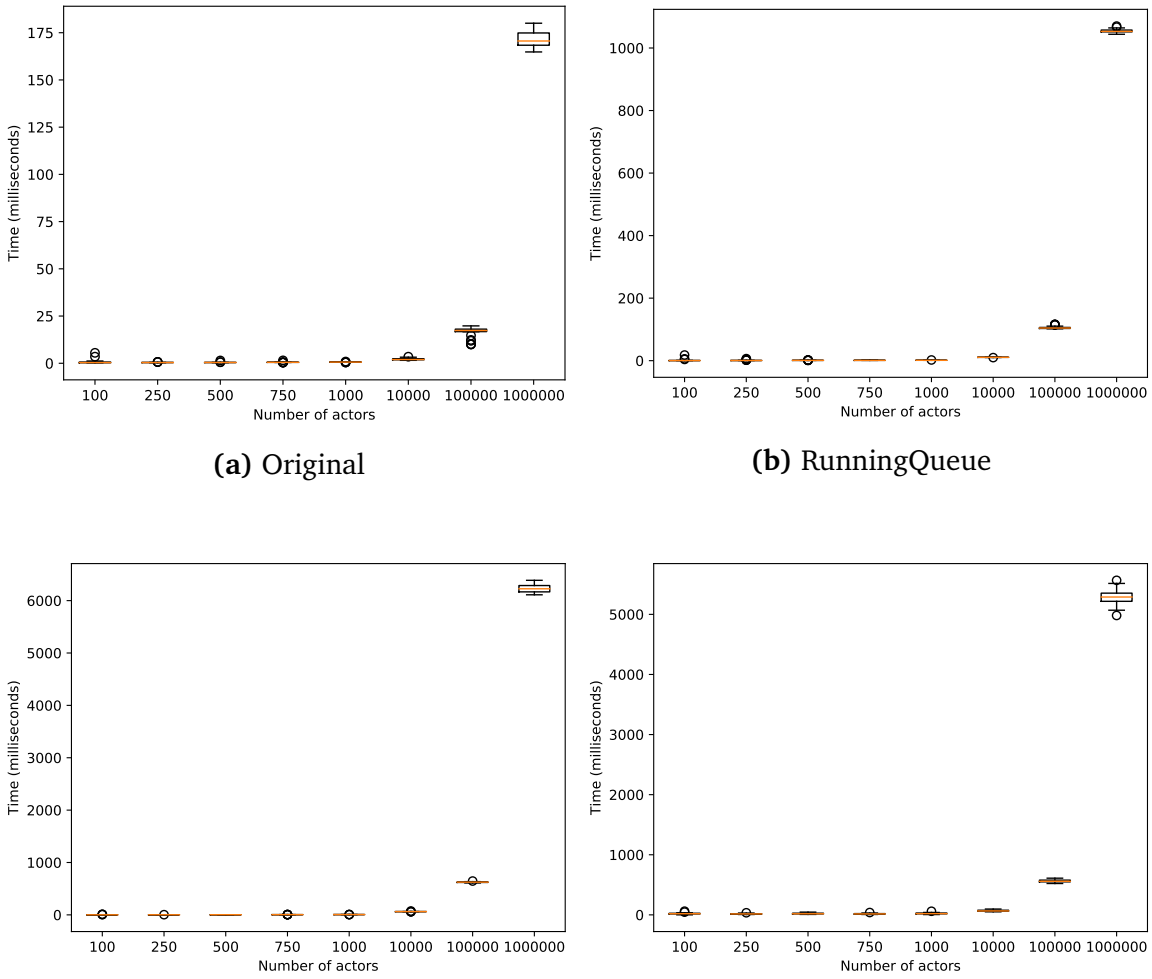


Figure 5.3: Counting Actor: time of execution against number of messages sent. Both scales are logarithmic.

Results analysis

Figure 5.3 shows the results for the different implementations. Immediately it can be noticed that the two implementations that faired worst in the previous benchmark now perform best. In particular **Original** is the fastest strategy for this benchmark. This might be due to the fact that there are only two processes in this system, each of which runs in a thread. Given that there are only two processes and the level of concurrency is low (unlike Chameneos) context switching between JVM threads is good enough.

While **WaitQueue** seems to initially perform better than **WaitQueueImproved**, for larger number of messages they converge to a reasonably equal performance. Once again, similarly to Chameneos, this may be due to **WaitQueue** double-scheduling InChannels. This is not a problem in terms of having to wait for a specific channel



(a) Original

(b) RunningQueue

(c) WaitQueue

(d) WaitQueueImproved

Figure 5.4: Performance results for the Counting Actor benchmark

to get to the front of the InChannels Queue, because given that there is only one process receiving, all the extra copies are actually of the same channel. However the negative effect that this larger queue size has on the speed of scheduling of InChannels due to the implementation details of LinkedTransferQueue described in the Result Analysis of Chameneos (Section 5.3.1), may affect the overall time of execution.

Regarding **RunningQueue**, the reason it performs better than **WaitQueue** and **WaitQueueImproved** might be because there are only maximum two processes in the Running Queue at any point in time, which likely makes the more sophisticated strategy by **WaitQueue** and **WaitQueueImproved** not worth having the extra InChannels Queue and the extra steps required to use it.

For this benchmark the variance (Figure 5.4) is very limited for all the implementations. This simply confirms that **Naive** is actually the best implementation for achieving large throughput in very small systems.

5.3.3 ForkJoin Creation

Overview

This benchmark, based on its homonym described in [13], specifically measures the cost of creating processes, by measuring the time required to have N processes ready to execute. In particular, we want to check how long it would take to spawn N processes in the worst scenario, which is when all the processes stay in memory for long enough for the spawning to finish before the first process finishes executing.

However as soon as the process system is started, the Executors and the InputExecutors start consuming processes. The only way to guarantee that this will be the case is to use `In` processes, because they will block and wait for a value to be sent to the channel they need to read. All other types would just immediately be executed by the Executor and there would be no guarantee (in fact it would be quite an unlikely occurrence if it happened) that all the processes are spawned before any process has finished executing.

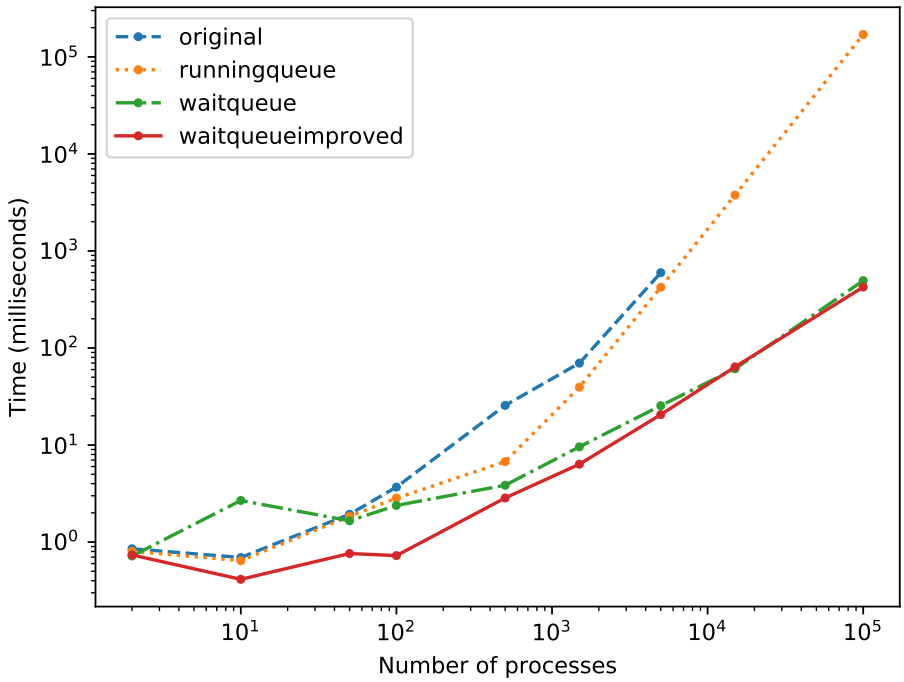


Figure 5.5: Fork Join Creation system: time of execution against number of processes created. Both scales are logarithmic.

Results analysis

Figure 5.5 shows the results for the different implementations. Predictably **Original** crashes, however this time is at around 10000 processes.

Interestingly the results for the optimised implementations are not the same, which would be reasonable to expect given that they all start by appending processes to the Running Queue.

This is likely due to the use of `In` processes in the implementation of the benchmark. In fact **WaitQueue** and **WaitQueueImproved** will remove those `In` processes from the Running Queue because they are not the responsibility of the Executors. While this may, in the case of **WaitQueue**, increase the size of the InChannels Queue, it would also leave less elements in the Running Queue. Since spawning relies on appending process states onto the Running Queue, if the queue has smaller average size the overall time required for spawning all the processes could be reduced, due to the implementation details of LinkedTransferQueue – on which the Running Queue is based – described in the Result analysis of Chameneos (Section 5.3.1).

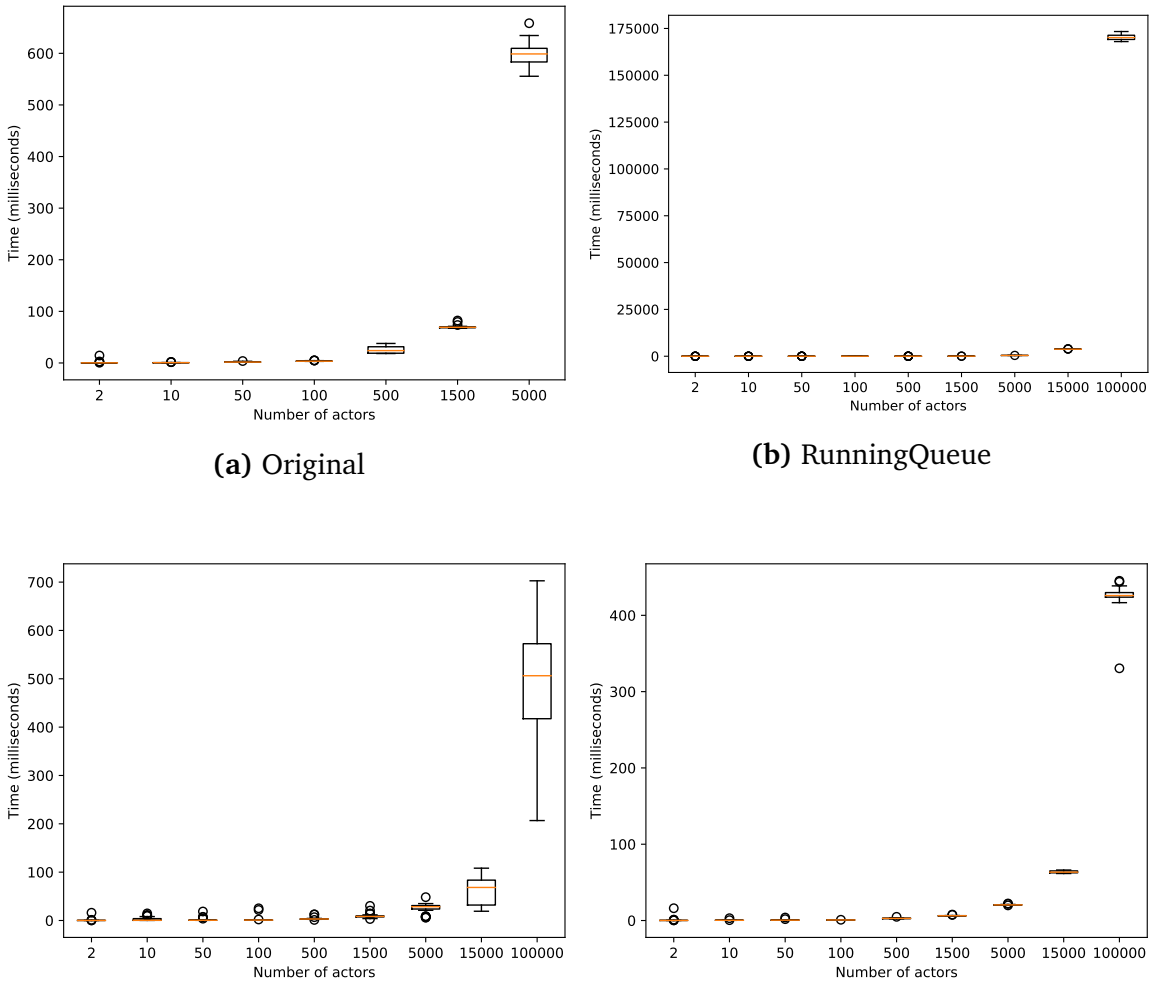
From the plots in Figure 5.6 the only implementation that seems to have an increasingly wider variance at larger scale seems to be **WaitQueue**. This is quite significant because in terms of average performance **WaitQueue** and **WaitQueueImproved** converge at higher scale, suggesting they are both valid implementations to use for large concurrent systems.

However this variance suggests that **WaitQueue**'s performance is much less predictable. We can see that **WaitQueueImproved** takes a fairly consistent 400 milliseconds to spawn 100000 processes, with only few sporadic outliers. **WaitQueue** on the other hand takes 200 milliseconds in the best case scenario, but can also realistically take up to 700 milliseconds. This means that **WaitQueueImproved** may be a better solution as it provides better worst case scenario performance.

5.3.4 Fork Join Throughput

Overview

This benchmark [13] aims to measure messaging throughput. Specifically, it ignores the time of creation (which should be measured by Fork Join Creation) and focuses on unidirectional messaging. There are M processes each receiving N messages. There is a single process dealing with sending all of the $M \times N$ messages, therefore the ability of processes to deal with concurrent incoming messages is not tested in this benchmark. Fork Join Throughput is vaguely similar to Counting Actor, in the sense that they both test throughput. However Counting Actor focused on messages, while this benchmark tests the effect of varying the number of processes.



(a) Original

(b) RunningQueue

(c) WaitQueue

(d) WaitQueueImproved

Figure 5.6: Performance results for the ForkJoin Creation benchmark

Results analysis

Figure 5.7 shows the results for the different implementations. The first thing worth noting is how significantly worse **Original** is compared to the optimised implementations, even in the range before crashing. Just like in Chameneos, this might be due to the high cost of context switching between JVM threads.

It is also interesting how, while in Chameneos **RunningQueue** was so slow **Original** caught up with it before crashing, in this benchmark it performs quite well, to the point that it catches up to **WaitQueueImproved** in terms of performance.

This could be due to the fact that in Chameneos each process performs both *sends* and *receives*, while here there is only one process doing all the sending and, more importantly, it does only sending. This means that even if a *receive* fails and that

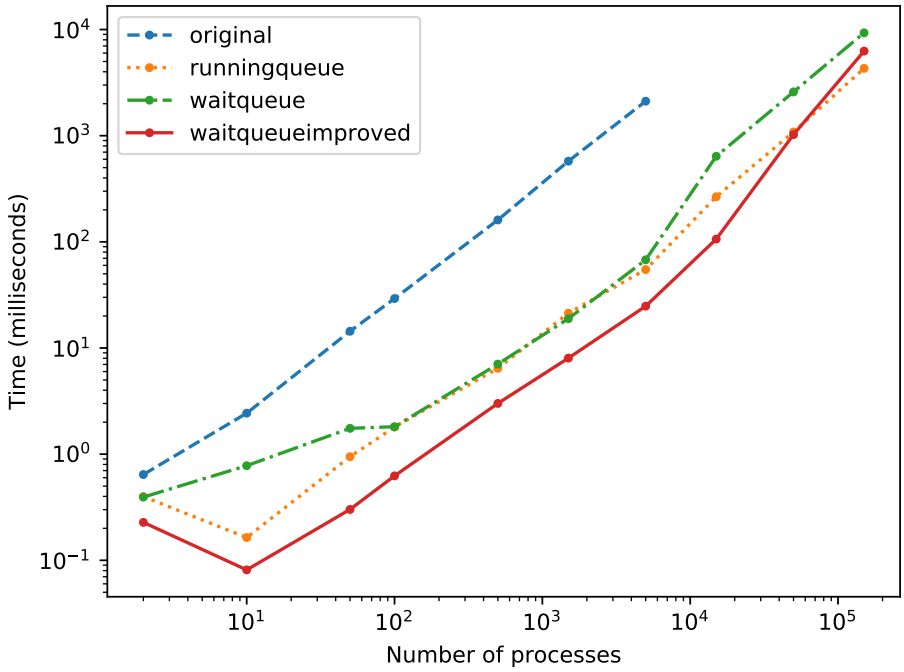


Figure 5.7: Fork Join Throughput system: time of execution against number of processes consuming the messages. Both scales are logarithmic. 200 messages are being sent to each process

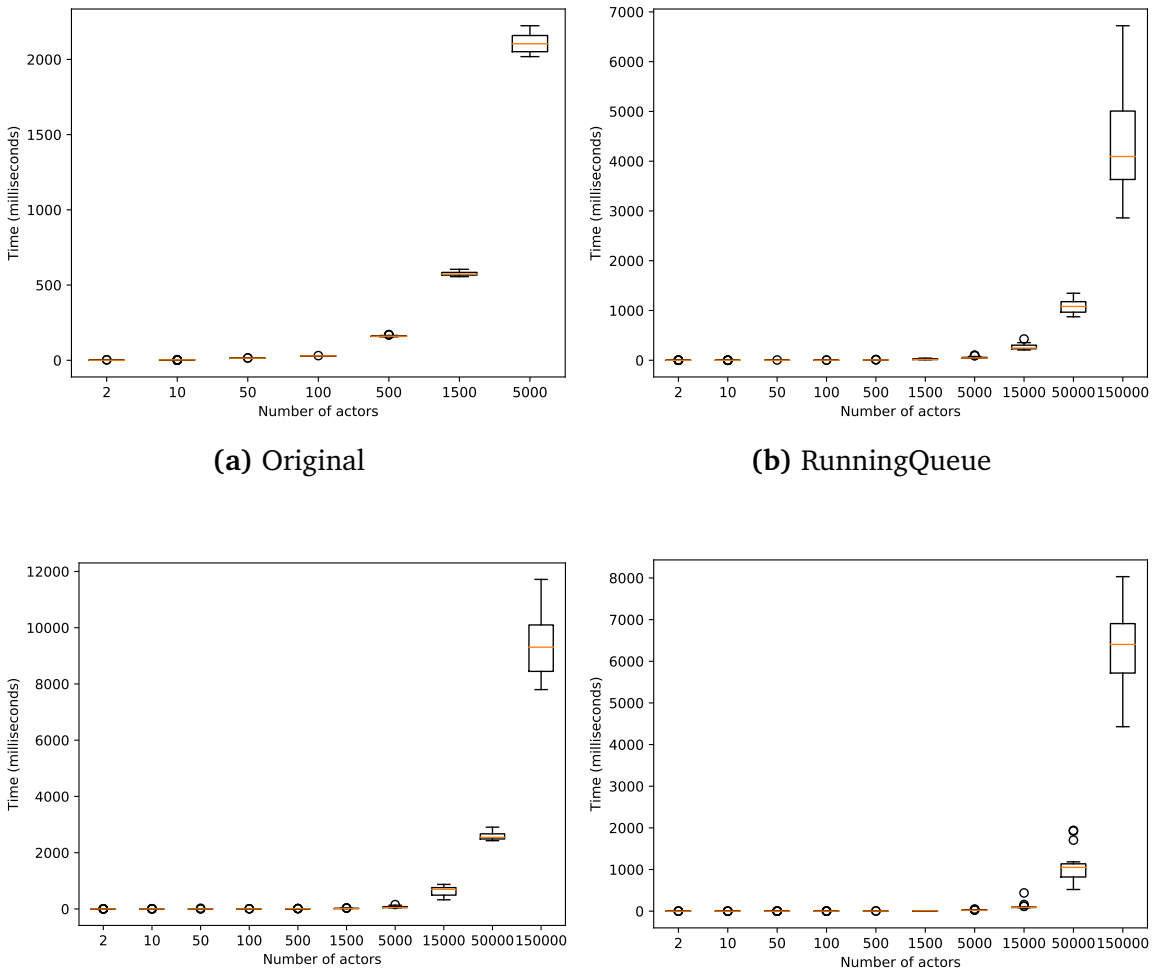
process is rescheduled, nothing depends on it and therefore the system can continue to advance its execution with other processes.

In fact, given that it does not have an extra queue and extra steps added by a more sophisticated logic like in **WaitQueue** and **WaitQueueImproved**, just like in Counting Actor, **RunningQueue** might even perform better at an even larger scale

Once again the slight gap in performance between **WaitQueue** and **WaitQueueImproved** may be due to the double-scheduling of InChannels slowing down scheduling for **WaitQueue**.

Analysing the plot of Figure 5.8 we notice that the only runtime implementation that does not have significant variance in its results at large scale is **Original**. However **Original**'s performance, even before crashing, is so significantly worse than that of all other implementations, that its stability is not sufficient to make it comparable.

RunningQueue, **WaitQueue** and **WaitQueueImproved** all seem to have increasingly larger variance as the system size grows. However **RunningQueue** has the best results with its running times ranging between 3000 and 6500 milliseconds, while the second best is **WaitQueueImproved** with a range of 4500 to 8000. From this results **RunningQueue** is the best implementation to handle unidirectional throughput in large concurrent systems.



(a) Original

(b) RunningQueue

(c) WaitQueue

(d) WaitQueueImproved

Figure 5.8: Performance results for the ForkJoin Throughput benchmark

5.3.5 Ping-Pong

Overview

The Ping-Pong system (based on [13] [8]) used in this benchmark is very similar to that used in the examples throughout the implementation (explained in Section 4.2.4 and described in Listing 3.1), with a pair of processes – *Ping* and *Pong* – which communicate back and forth. However in this case the pairs communicate back and forth for 200 times, and what is being tested is the time of execution as the number of independent pairs in the system increases.

While this benchmark bears some similarity to Fork Join Throughput its goal is different. It focuses more on the round-trip time required for processes to send a message and receive a response. Furthermore, unlike Fork Join Throughput, for each process

the *sends* are intertwined with *receives*, and the sends are also performed in parallel by different processes.

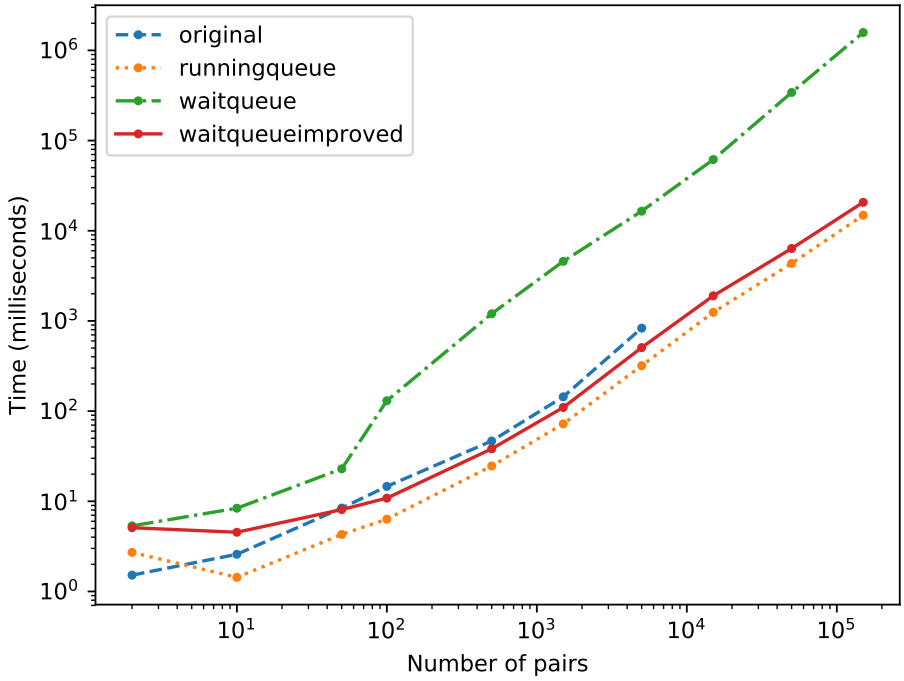


Figure 5.9: PingPong system: time of execution against number of pairs. Both scales are logarithmic

Results analysis

Figure 5.9 shows the results for the different implementations. Once again **Original** crashes at around 5000. This is because what is recorded is the number of pairs, and each has a *ping* and a *pong* process.

RunningQueue performs not just as well but slightly better than **WaitQueueImproved**.

In general, this good performance could be due to the pairs being are independent of each other and therefore even if there is a failed *receive* that causes an `In` process to be put to the back of the Running Queue, the system's overall execution can continue.

Specifically, the reason **RunningQueue** is slightly faster than **WaitQueueImproved** may be due to the way multi-step execution is implemented. In **RunningQueue** both *sends* and *receives* are executed by the Executor, which means that it can fully benefit from the multi-step nature of `fastEval` and perform up the maximum number of steps (unless due to thread scheduling the receive is delayed). However in **WaitQueue** and **WaitQueueImproved** when `fastEval` encounters and `In` process this is

not executed directly, it is only set to be eventually processed by the InputExecutors when its channel is scheduled. And when it is, after `inEval` executes the *receive*, since the next process is a send, this is not immediately executed but it is scheduled to be consumed by the Executors. This essentially robs these two implementations of all the benefits of a multi-step evaluation. A potential solution is discussed in Future Work (Section 6.2).

The even worse performance of **WaitQueue** could be once again due to the double-scheduling of `InChannels`, which increase the average size of the InChannels Queue. If there are many copies of a specific InChannel c scheduled, but an `Out` sending to it is at the back of the Running Queue, this will likely slow down the InputExecutors.

The variance of the results for each implementation is shown in Figure 5.10. In this case it seems that all implementations have reasonably consistent results, with only slight increase in variance at larger scale. This confirms that **RunningQueue** is the most apt solution for concurrent systems similar to Ping-Pong, with many independent components requiring fast round trip time in internal communication.

5.3.6 Thread Ring

Overview

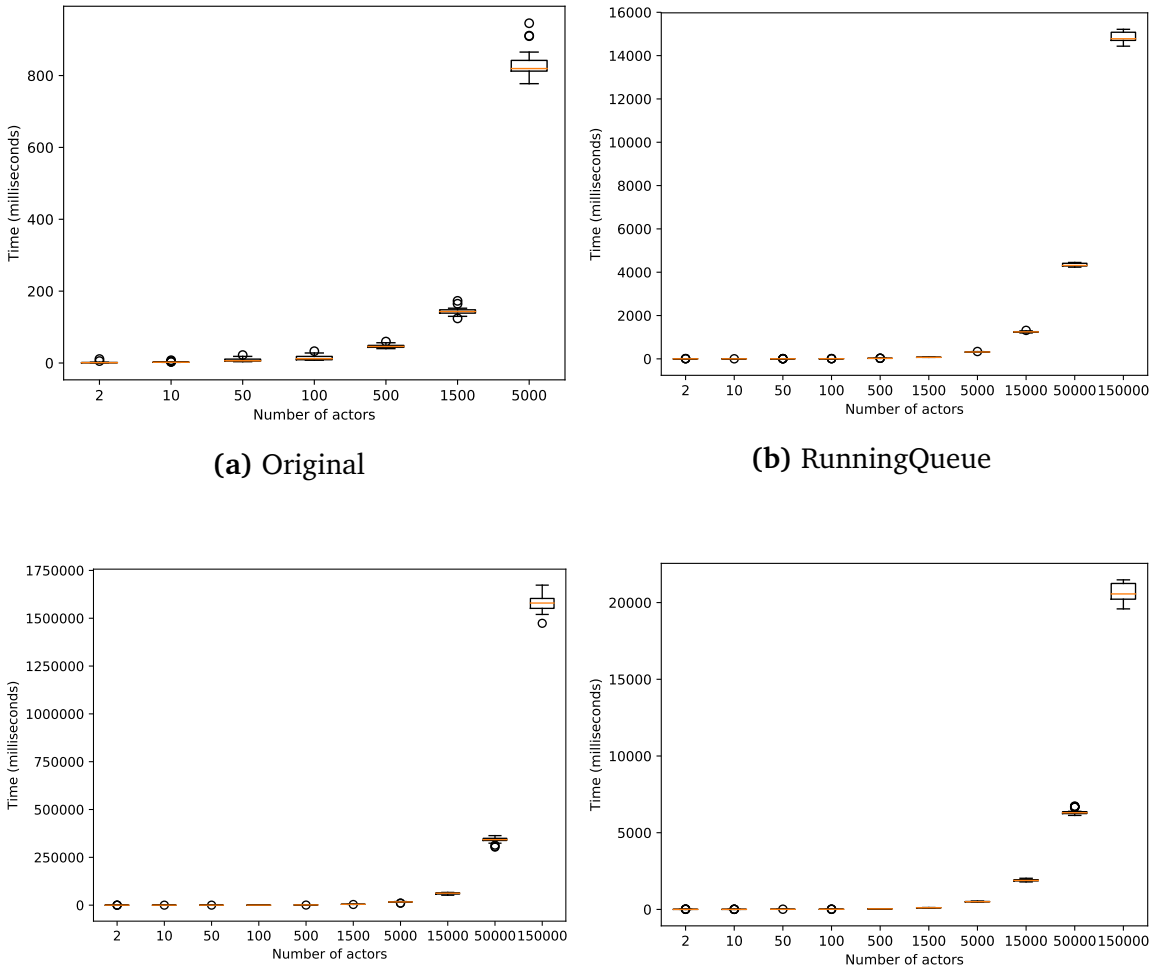
This benchmark [13] [2] instantiate a ring like structure where a token message is passed around from member to member until the maximum number of hops allowed is reached. Notably this benchmark's execution is actually sequential regardless of how many members (processes) there are in the ring. Its goal is to test the ability of the different implementations to efficiently context switch between processes while most of the system is hung waiting for a message.

Results analysis

Figure 5.11 shows the results for the different implementations. The first notable detail of this plot is that right until it crashes due to too many threads being spawned, **Original** seems to have the best performance out of all. This is probably because when so many JVM are spawned, context switching becomes extremely expensive.

However it seems safe to assume that, considering scalability, the best implementation seems to be **WaitQueueImproved**. While it starts worse than all other implementations, it actually seems to have constant complexity, remaining fairly unchanged regardless the number of ring members.

On the other hand both **WaitQueue** and **RunningQueue** seem to have linear complexity with regard to the number of ring members. This is an extremely significant



(a) Original

(b) RunningQueue

(c) WaitQueue

(d) WaitQueueImproved

Figure 5.10: Performance results for the Ping-Pong benchmark

difference, and it is likely due to the way the three optimised implementations deal with `In` processes. Upon failure in receiving, **RunningQueue** reschedules an `In` process to the back of the Running Queue. This means that even if immediately after a value arrives on the channel that process needs to read, the whole queue of processes will have to be consumed before it gets a chance. Similarly in **WaitQueue**, when the processes are spawned (all of which but one are `Ins`) the first thing that happens is that all their InChannels are scheduled, filling up the InChannels Queue, which now is slowly consumed in order. However in **WaitQueueImproved** the scheduling of `InChannels` is only done when a *send* is executed. This means that all the `In` processes will not be stored in either queue, but will be dormant on their respective InChannel's Wait Queue, and therefore the scheduling will be much faster.

The only implementation whose performance results have considerable variance (Figure 5.12) is **WaitQueueImproved**. However in Figure 5.11 we saw that its performance is considerably better than any of the other implementation. We see that for 50000 processes it takes it between 320 and 360 milliseconds to run (with a

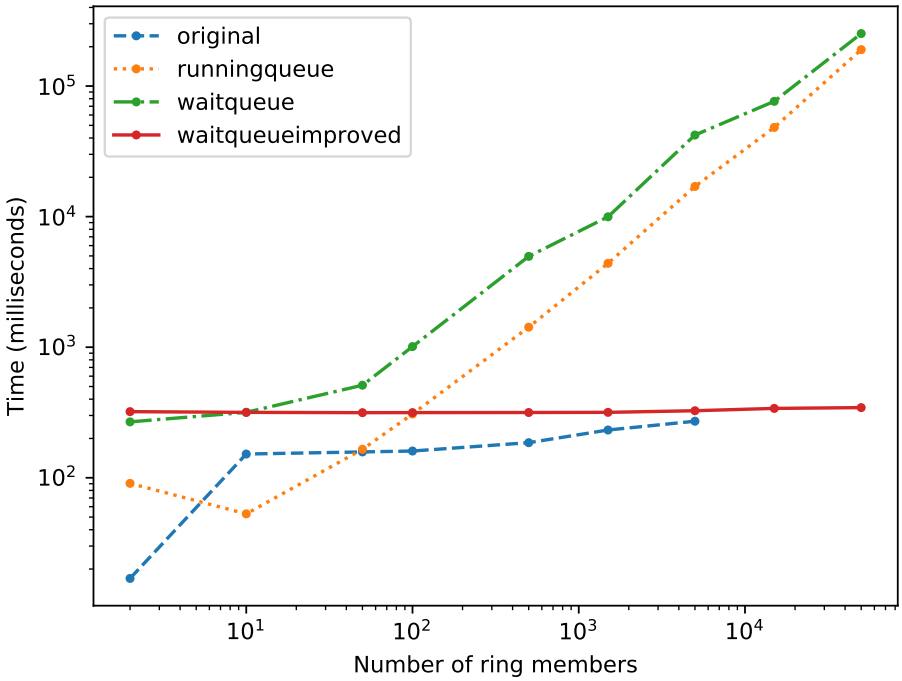


Figure 5.11: Thread Ring system: time of execution against number of ring members. Both scales are logarithmic

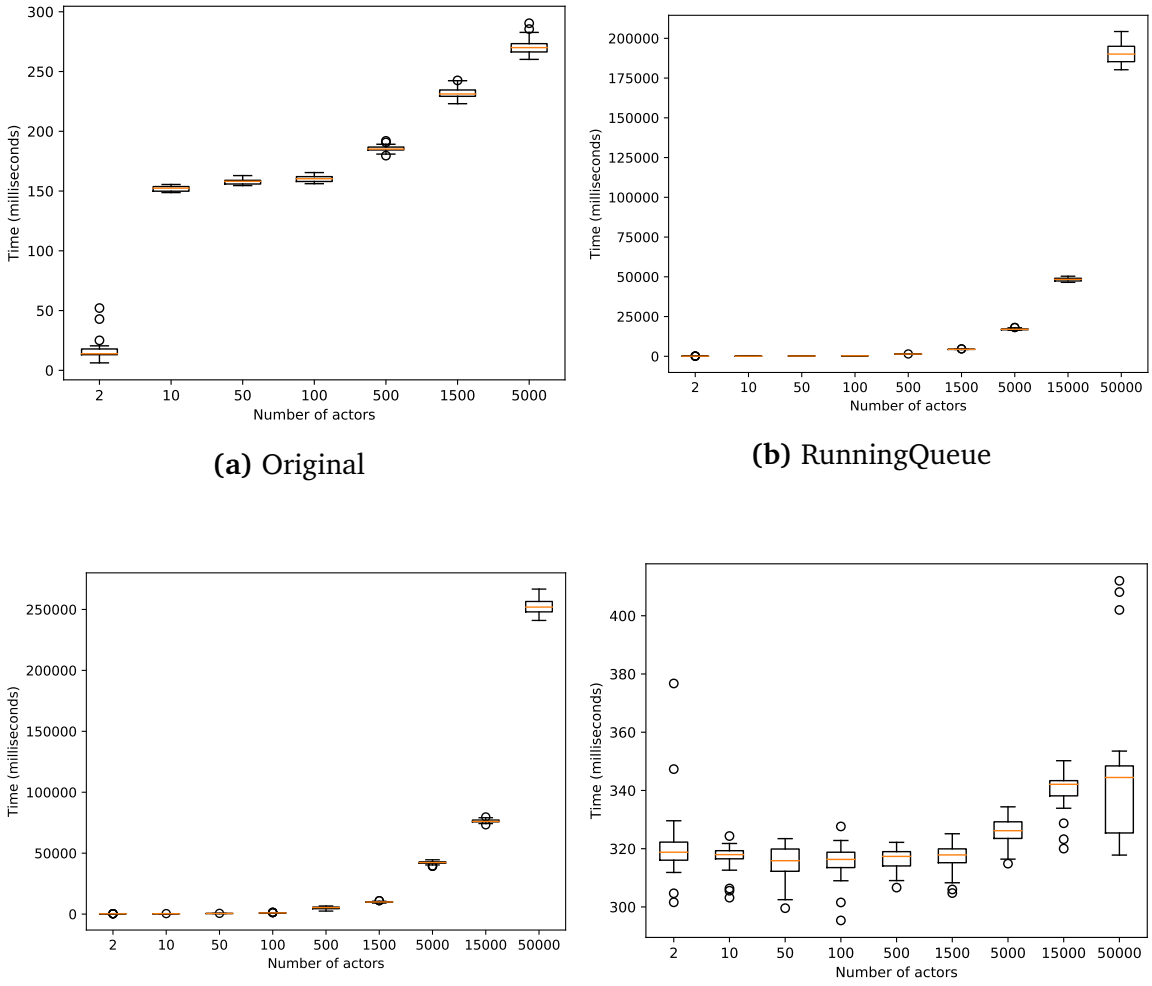
few outliers at around 410 milliseconds). However the second most efficient implementation, **RunningQueue** takes around 190000 milliseconds to run, which is worse by a factor of $\times 1000$.

5.4 Fine tuning the default number of threads per core

The number of threads per core (which in turn determines the number of Executors and InputExecutors) is a significant parameter to the library, which can affect its performance. While we give the option to modify this parameter to cater for all possible use cases, it is valuable to provide a sensible default value, which should be optimal for the most common scenarios. In this section we explore the performance of each runtime implementation introduced in Section 4 with varying number of threads per core. We will proceed to reason by implementation, as a different default value can be set for each if needed.

RunningQueue

Figure 5.13 shows that for Counting Actor and Ping-Pong, there is some variance in performance using a single threads. However all other benchmarks' results indicate



(a) Original

(b) RunningQueue

(c) WaitQueue

(d) WaitQueueImproved

Figure 5.12: Performance results for the Thread Ring benchmark

that performance decreases with the number of threads per core, and so does the variance – and therefore the unreliability – of the results.

The ideal value that seems to yield good performance and low variance in all benchmarks is 2 threads per physical core.

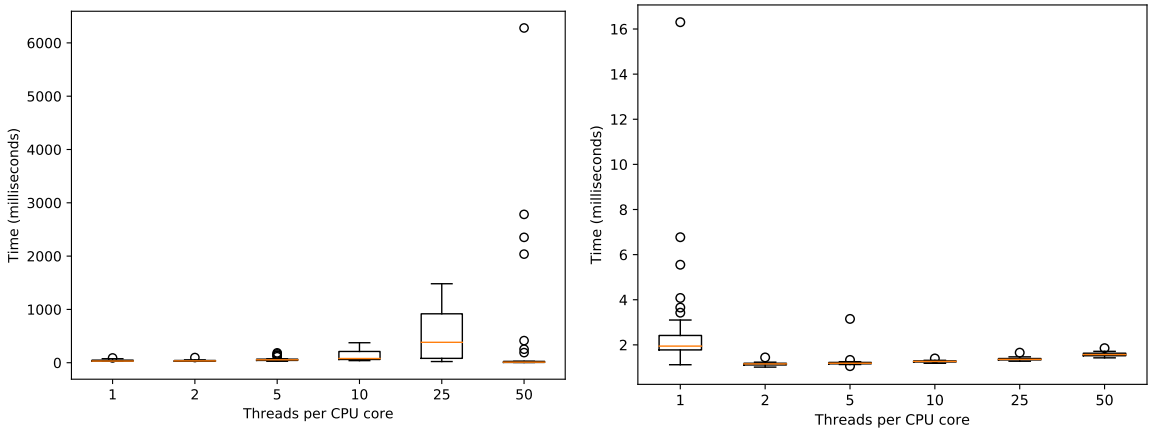
WaitQueue

Figure 5.14 shows that for Counting Actor and Ping-Pong, there is some variance in performance using a single threads. However all other benchmarks' results indicate that performance decreases with the number of threads per core, and so does the variance – and therefore the unreliability – of the results.

The ideal value that seems to yield good performance and low variance in all benchmarks is 2 threads per physical core.

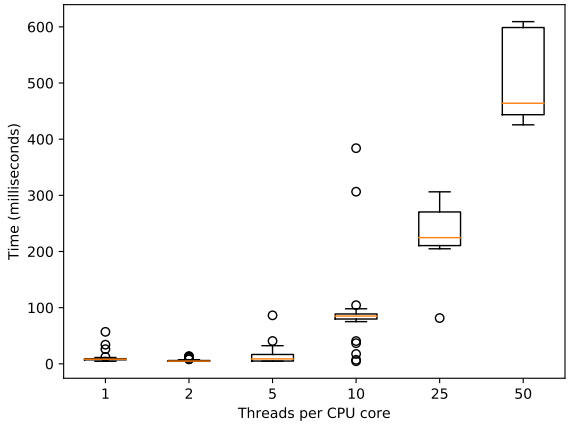
WaitQueueImproved

The results are shown in Figure 5.15. ForkJoin Throughput seems to always have some variance in the results, but especially with a single thread per core, which also yields the worst performance. ForkJoin Creation has less variance in performance, but one thread per core still yields the worst performance. The results from Chameneos, Counting Actor and Ping-Pong show that performance decreases with the number of threads per core. Thread Ring follows a similar trend, however a single thread per core also yields to performance. Once again the ideal value seems to be around 2 threads per physical core.

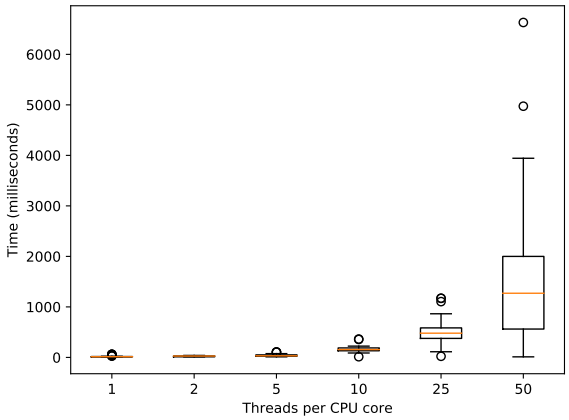


(a) Chameneos

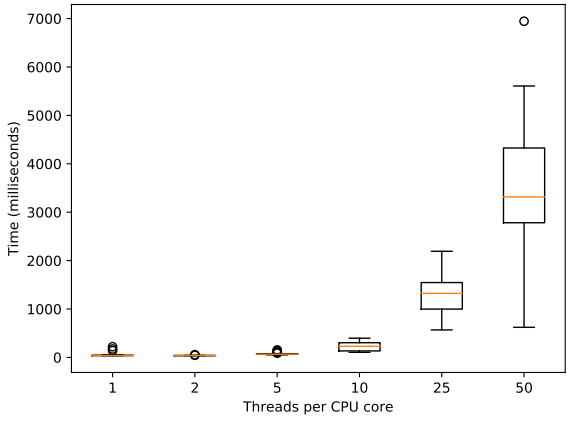
(b) Counting Actor



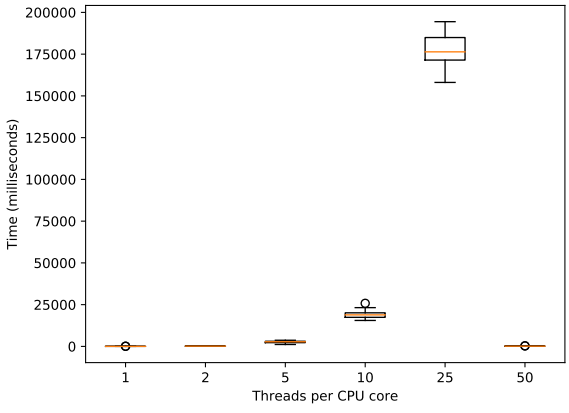
(c) Fork Join Creation



(d) Fork Join Throughput

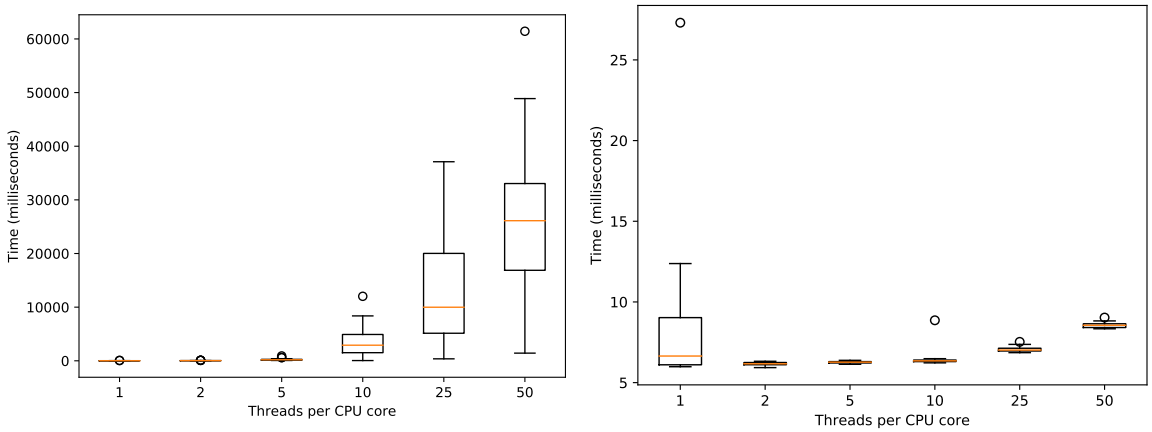


(e) Ping-Pong



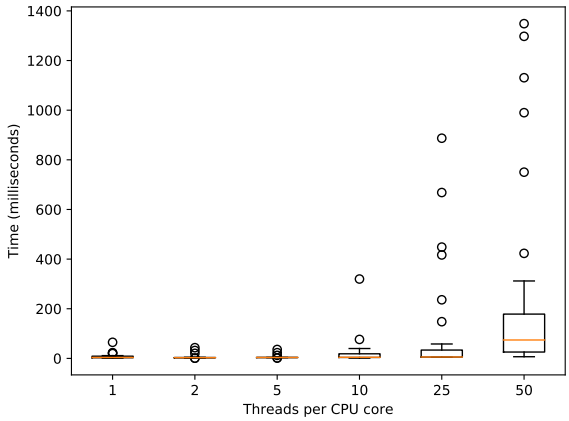
(f) Thread Ring

Figure 5.13: RunningQueue results varying number of threads per core

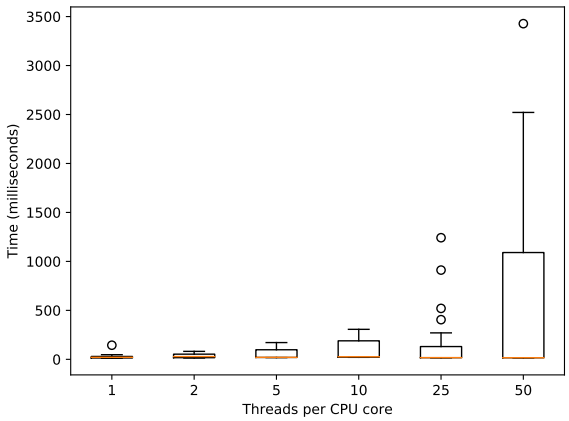


(a) Chameneos

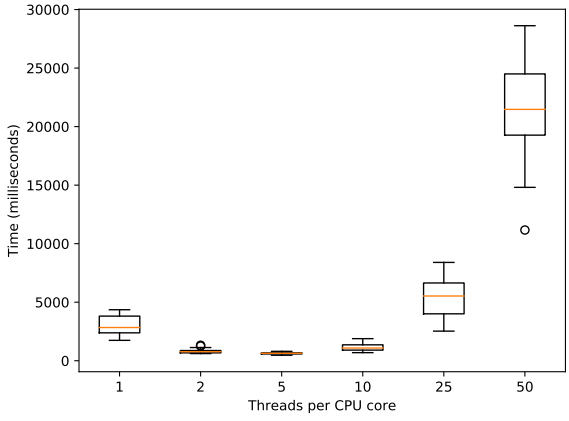
(b) Counting Actor



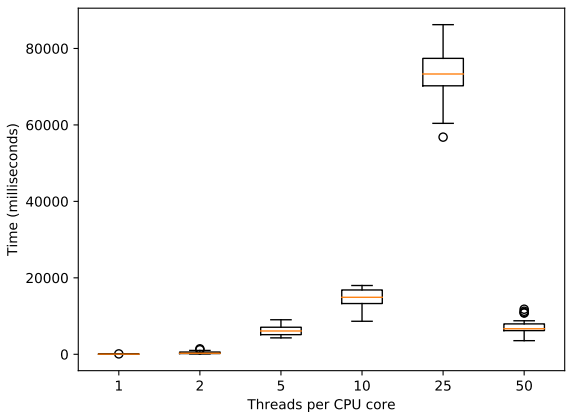
(c) Fork Join Creation



(d) Fork Join Throughput

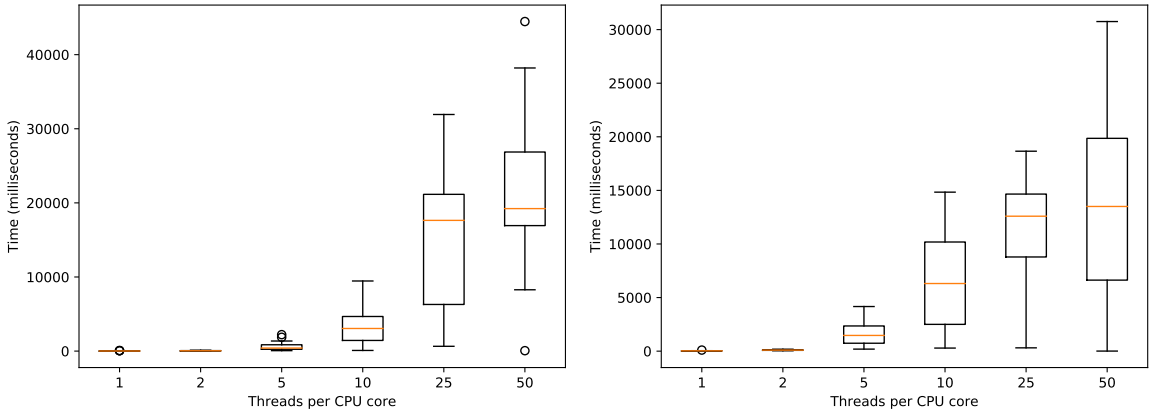


(e) Ping-Pong



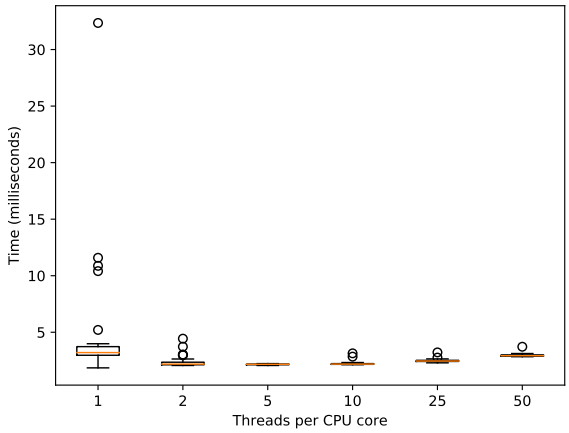
(f) Thread Ring

Figure 5.14: WaitQueue results varying number of threads per core

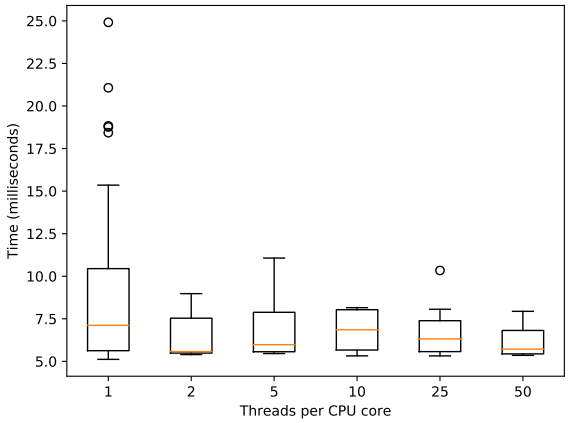


(a) Chameneos

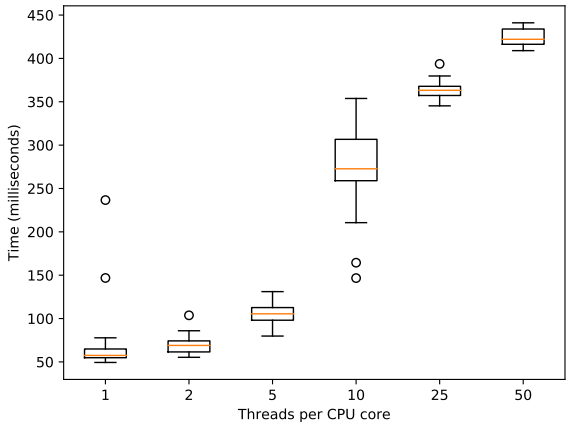
(b) Counting Actor



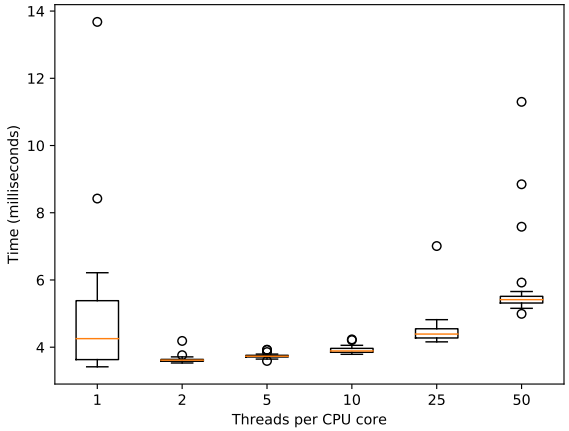
(c) Fork Join Creation



(d) Fork Join Throughput



(e) Ping-Pong



(f) Thread Ring

Figure 5.15: WaitQueueImproved results varying number of threads per core

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We now discuss the outcome of the project relating it to the objective listed in the Introduction (Section 1.2).

Objective 1: Preserving correctness

No change was required to the DSLs nor their types, and therefore the new runtime designs provide the same correctness guarantees as the original implementation.

The only changes made that surfaced to the public API level is that before being able to start spawning processes, it is necessary to instantiate a Process System and that in order to terminate the execution of a program, the `kill` method of the Process System must be explicitly called. We believe this is an acceptable change given that, as mentioned in Section 4.2 where the Process System is introduced, this resembles Akka's Actor System, which is also generally explicitly instantiated.

Objective 2: Scalability

We achieved the first fundamental objective of this project – allow Effpi to run large concurrent systems without the program becoming non-responsive – by decoupling effpi-processes from JVM threads (Section 4.2). As evident from the benchmarks results shown in the Evaluation in Section 5.3, all of our runtime implementations have improved on that aspect compared the original version. It can be clearly seen that while the original runtime crashes at around 10k processes, our implementations still work successfully at 50k-100k processes, and more.

Objective 3: Performance

We improved speed of execution with respect to the original runtime by minimising bookkeeping operations. In Section 4.3 and Section 4.4 we managed to make improvements to the overall performance by reducing the impact of scheduling input-processes not ready to be executed. We did so having two separate scheduling systems for input-processes and for all other types of processes. This solution meant that while scheduling input-processes not ready to execute can still slow down other input-processes, the effect on processes of other types is alleviated.

Further success was achieved in Section 4.5 by reducing the relative time spent in the execution performing context switching between processes. This is done by allowing Executors and InputExecutors to perform multiple steps of evaluation of a process before rescheduling it.

The results discussed in Section 5.3 show how we achieved an increase of speed of execution of up to a factor of 100 (e.g. in the ForkJoin Throughput benchmark).

Objective 4: Customisability

We allow the users to customise the number of threads created per physical core of the host machine, because depending on the use case, they might benefit from tweaking that value. However we fine-tuned its default value to 2 as a consequence of benchmarking, discussed in Section 5.4.

6.2 Future Work

Reducing the number of failed receives by avoiding blind rescheduling

This is a very important area of future improvement. Essentially the main goal would be to avoid blind rescheduling inside the InputExecutor without incurring in the deadlock described in Section 4.3.9

A potential solution (once again inspired by the implementation of Akka's Mailboxes [20]) involves using a state-machine to describe the state of each InChannel, to capture the following three states: unscheduled, scheduled and running (scheduled but currently being considered in the execution of an InputExecutor). Essentially this would capture the deadlock situation just mentioned and, only in that specific case, issue a rescheduling of the InChannel.

Keeping this state would also make possible to schedule only one reference to each channel at a time. This would minimise the size of the InChannels Queue. Since

this is internally represented as a LinkedTransferQueue, due to the latter's implementation details described in the Result Analysis of the Chameneos benchmark in Section 5.3.1, this could lead to better performance of scheduling.

Improve multi-step execution

In Section 5.3 we analysed how the current implementation is not optimal with systems such as Ping-Pong, where a process performs *receives* alternated to any other type of process. This is because it requires to keep moving the process from the InputExecutors to the Executors, as now the scheduling responsibilities are fully separated.

One solution could be to maintaining the scheduling of input processes separated, but to provide both Executors and InputExecutors with the logic to evaluate all process types. This for example would allow an Executor that encounters an input process to evaluate it if ready to run, otherwise leaving it to the InputExecutors. Similarly InputExecutors would be able able to execute any non-input process that is part of the continuation of a scheduled input-process up to the maximum number of consecutive steps. This would require to change the evaluation functions `fastEval` and `inEval` to contain some overlapping logic.

Smart management of Executors and InputExecutors

A less clearly defined improvement could involve the instantiation of Executors and InputExecutors. At the moment they are statically instantiated in fixed number, however this is not necessarily always the best solution.

First of all it is unlikely that spawning the same number of Executors as InputExecutors is always the optimal strategy. Perhaps an analytical tool that parses the process data structures and tries to guess the optimal Executors to InputExecutors ratio could lead to better runtime performance.

Secondly there might be systems where, for example, at different times of execution there are more (or less) input-processes. In such a scenario having Executors and InputExecutors instantiated and terminated dynamically to optimise resource usage might lead to better performance. This would probably require a runtime monitoring system of the state of the Running Queue and InChannels Queue to establish which type of scheduling system is under the most stress, and which could potentially yield some spare capacity.

Akka Integration

Given the spread of Akka and how well established it has become in industry, it would be extremely beneficial for Effpi's adoption to allow for integrated use. This would enable systems of processes created with Effpi to intercommunicate with systems of Akka actors, thus immediately widening the number of existing projects that Effpi could interact with.

At least an initial integration (the ability to send and receive messages from actor to process and vice-versa) should not be too challenging. However some changes would be required due to the addition made during this project to the OutChannel trait, which at the moment requires a `dualIn` method, which Akka actors would not be able to provide since actors do not have the concept of dual, as they are not based on π-calculus and therefore do not use channels.

Formal proof of correctness of the new runtimes

Given the scope of this project, correctness was tested through testing and benchmarking. This strategy was sufficient to spot the issues with the two abandoned designs mentioned in Section 5.2, and at the same time no issue surfaced for the optimised implementation adopted. However this does not provide a formal guarantee of correctness, which could be a potential future area of study.

Providing the ability to set time outs

A feature with great practical value is the possibility to set time outs on the maximum time given to input-processes to receive a value. These are often used in practice for sanity check on systems and, since the original runtime provided that feature, it would be valuable to enable their use in the new runtime implementations too.

However since processes are now decoupled from JVM threads and their execution is often suspended, a smarter strategy to keep track of the waiting times for each time out would have to be introduced, potentially dedicating a whole thread for this. To this purposes we could, once again, take Akka's approach as inspiration.

Further scalability testing

Since the current benchmarks aimed at comparing the different implementations, the size of the systems attempted never exceeded the hundreds of thousands of concurrent agents. In order to asses the current limits on scalability testing should be done with significantly larger benchmarks. This will likely require longer running

times and more resources. In particular it might be necessary to run the JVM with custom parameters, such as more heap space.

Fine-tuning the default value for maximum number of consecutive evaluation steps

While this was beyond the scope of the project – and therefore the reasonable, but somewhat arbitrary default value of 10 was chosen – it might be beneficial to fine-tune this default value like it was done for the number of threads per physical core. This is actually non-trivial, because in order to have a meaningful quantitative analysis a large number of benchmarks would be needed, presenting different system sizes and different granularity requirements. For the time being the ability to customise this value is provided.

Bibliography

[1] A. Mason. The countingactor benchmark, 2018. http://www.theron-library.com/index.php?t=page&p=countingactor. pages 73

[2] A. Mason. The threadring benchmark, 2018. http://www.theron-library.com/index.php?t=page&p=threadring. pages 81

[3] ABCD. A basis for concurrency and distribution, 2018. pages 7, 10

[4] Ahmad Elshareif. Which companies use the Erlang language?, 2018-06-10. https://www.quora.com/Which-companies-use-the-Erlang-language. pages 1

[5] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theoretical Computer Science*, 669:33 – 58, 2017. pages 6, 11

[6] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015. pages 6, 9, 10, 11

[7] Doug Lea with assistance from members of JCP JSR-166 Expert Group. Linkedtransferqueue, 2018-06-10. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/LinkedTransferQueue.java?view=co. pages 71

[8] École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. pingpong.scala, 2018. https://www.scala-lang.org/old/node/54. pages 79

[9] golang.org. The go blog - share memory by communicating, 2018-06-9. https://blog.golang.org/share-memory-by-communicating. pages 1

[10] High Scalability. Product: Amazon's simpledb, 2018-06-10. http://highscalability.com/product-amazons-simpledb. pages 1

[11] High Scalability. The whatsapp architecture facebook bought for 19 billion, 2018-06-10. http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html. pages 1

[12] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998. pages 11

[13] Shams M. Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80, New York, NY, USA, 2014. ACM. pages 13, 69, 73, 75, 76, 79, 81

[14] C. Kaiser and J. F. Pradat-Peyre. Chameneos, a concurrency game for java, ada and others. In *ACS/IEEE International Conference on Computer Systems and Applications, 2003. Book of Abstracts.*, pages 62–, July 2003. pages 69

[15] Lightbend, Inc. Actor Systems, 2018-05-3. `https://doc.akka.io/docs/akka/2.5/general/actor-systems.html`. pages 46, 68

[16] Lightbend, Inc. Akka, 2018-06-9. `https://akka.io/`. pages 1

[17] Lightbend, Inc. Dispatchers, 2018-06-9. `https://doc.akka.io/docs/akka/2.5/dispatchers.html`. pages 46

[18] Lightbend, Inc. Dispatcher.scala, 2018-06-9. `https://github.com/akka/akka/blob/master/akka-actor/src/main/scala/akka/dispatch/Dispatcher.scala`. pages 46

[19] Lightbend, Inc. Mailboxes, 2018-06-9. `https://doc.akka.io/docs/akka/2.5/mailboxes.html`. pages 46

[20] Lightbend, Inc. Mailbox.scala, 2018-06-9. `https://github.com/akka/akka/blob/master/akka-actor/src/main/scala/akka/dispatch/Mailbox.scala`. pages 46, 90

[21] Martin Odersky et al. Dotty - A next generation compiler for Scala, 2018-06-13. `http://dotty.epfl.ch/`. pages 2

[22] Pivotal. pages 1

[23] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003. pages 12

[24] Alceste Scalas and Nobuko Yoshida. Dependent function types for higher-order interaction, 2018-06-10. `https://www.doc.ic.ac.uk/~ascalas/effpi/tech-report.pdf`. pages 1, 3, 13, 14, 15, 16, 19, 20, 22, 30

[25] The Erlang Language. Erlang - build massively scalable soft real-time systems, 2018-06-9. `https://www.erlang.org/`. pages 1

[26] The Pony Language. Pony, 2018-06-9. `https://www.ponylang.org/`. pages 1

[27] Haskell Wiki. Embedded domain specific language, 2018-06-13. `https://wiki.haskell.org/Embedded_domain_specific_language`. pages 2, 16

[28] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73 – 93, 2007. Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006). pages 11

Appendices

Appendix A

Chameneos Benchmark full code

Listing A.1: Effpi implementation of the full Chameneos benchmark

```
object Chameneos {

  implicit val timeout: Duration = Duration.Inf

  // Convenience types for the messages exchanged ====================

  enum Colour {
    case Blue
    case Red
    case Yellow
  }

  case class Request(replyTo: ActorRef[Response], mate:
      ActorRef[PtoPCommunication])

  enum Response {
    case Mate(replyTo: ActorRef[PtoPCommunication])
    case Stop
  }

  enum PtoPCommunication {
    case Col(colour: Colour)
    case Stop
  }

  // Chameneos System ================================================

  type Broker = Rec[RecAt, Read[Request, (Read[Request,
    SendTo[ActorRef[Response], Response] >>:
    SendTo[ActorRef[Response], Response] >>:
    Loop[RecAt]] | SendTo[ActorRef[Response], Response] >>:
```

```
31      (Loop[RecAt] | PNil))]]
32
33    // DSL expression describing the behaviour of the broker
34    def broker
35    (maxMeetings: Int, numChameneos: Int)
36    (startTimeFuture: Future[Long], endTimePromise: Promise[Long]) =
37      Behavior[Request, Broker] {
38      var meetings = 0
39      var stoppedChameneos = 0
40      prec(RecA) {
41        Await.result(startTimeFuture, Duration.Inf)
42        read {
43          case Request(replyToA, mateA) =>
44            if (meetings < maxMeetings) {
45              read {
46                case Request(replyToB, mateB) =>
47                  send(replyToA, Response.Mate(mateB)) >> {
48                    send(replyToB, Response.Mate(mateA)) >> {
49                      meetings += 1
50                      ploop(RecA)
51                    }
52                  }
53              }
54            } else {
55              if (stoppedChameneos == 0) {
56                endTimePromise.success(System.nanoTime())
57              }
58              send(replyToA, Response.Stop) >> {
59                stoppedChameneos += 1
60                if (stoppedChameneos < numChameneos) {
61                  ploop(RecA)
62                } else {
63                  nil
64                }
65              }
66            }
67        }
68      }
69    }
70
71    type Chameneos = Rec[RecAt, Spawn[PtoPCommunication,
72      PtoPChameneos, SendTo[ActorRef[Request], Request] >>:
73      Read[Response, ((SendTo[ActorRef[PtoPCommunication], PtoPCommunication]
          >>:
74      Loop[RecAt]) | SendTo[ActorRef[PtoPCommunication],
          PtoPCommunication])]]]
75
76    type PtoPChameneos = Read[PtoPCommunication, PNil]
```

```
77
78    // DSL expression describing the behaviour of a chameneos
79    def chameneos(broker: ActorRef[Request])(initColour: Colour) =
80      Behavior[Response, Chameneos] {
81        var colour = initColour
82        // println(s"Initial colour for $self is $colour")
83        prec(RecA) {
84          spawn (Behavior[PtoPCommunication, PtoPChameneos]{
85            read {
86              case PtoPCommunication.Col(matesColour) =>
87                colour = mutatedColour(colour, matesColour)
88                // println(s"The new colour for $self is $colour")
89                nil
90              case PtoPCommunication.Stop =>
91                nil
92            }
93          }) { ref =>
94            send(broker, Request(self, ref)) >>
95            read {
96              case Response.Mate(mate) =>
97                send(mate, PtoPCommunication.Col(colour)) >>
98                  ploop(RecA)
99              case Response.Stop =>
100               send(ref, PtoPCommunication.Stop)
101           }
102         }
103       }
104     }
105
106   // This function handles the chameneos colour mutation
107   private def mutatedColour(colourA: Colour, colourB: Colour) = {
108     if (colourA != colourB) {
109       (colourA.enumTag + colourB.enumTag) match {
110         case 1 => Colour.enumValue(2)
111         case 2 => Colour.enumValue(1)
112         case 3 => Colour.enumValue(0)
113       }
114     } else {
115       colourA
116     }
117   }
118 }
```