VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Engineering and Computer Science
Te Kura Matai Pukaha, Purorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Integrating the Actor Model into the Whiley Programming Language

Timothy Jones

Supervisors: Dr. David Pearce, Dr. Lindsay Groves

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering.

Abstract

Introducing concurrent behaviour into a program tends to also introduce unpredictable behaviour. The actor model is an attempt to simplify concurrency and reduce such problems. The Whiley programming language is structured around concurrent processes, which are analogous to actors, but it runs on top of the Java Virtual Machine, which does not provide support for the model. This project implemented the model for JVM in a manner specific to Whiley.

Acknowledgements

- David, for guiding me through the project and putting up with my chronic disorganisation and lateness.
- Lindsay and James, for relaxing deadlines.
- Unlimited Realities, and particularly Russell, for lending a laptop for the whole year.
- David Brebner, for sparking my interest in concurrent programs with his own ideas.
- Jeremy Ashkenas, for sparking my interest in general language design.
- Chris, Dan and Fahmi, for listening to my rants.
- Richard, for maintaining deadlines with threats (and then forgetting).
- Emma, for providing advice on when to sleep.
- Reddit, for endless distractions that kept me sane.

Contents

Chapter 1

Introduction

Concurrency has always been a difficult problem, but only in recent years, as multicore processors have come to the forefront, have they risen to prominence. The most common and basic form of concurrency is the use of threading, but this doesn't regulate the way that concurrent objects interact with one another and tends to lead to exceptional behaviour. The actor model is an attempt to more closely regulate these interactions and avoid many possible problems in the process.

Many programming languages already have support for the actor model, most notably Erlang [2] and Scala [8]. The fledgling Whiley language uses it as well, but its current implementation is sorely lacking. It easily surpasses reasonable memory and performance limits, and needs a new implementation that does not suffer from these problems.

This is not a simple task. Whiley's runtime environment is the Java Virtual Machine, which only supports heavy threading natively. The actor model demands lightweight concurrent primitives, so mapping them directly onto these heavy threads is unacceptable — this is what the current implementation does. Instead, the lightweight actors need to be distributed across a fixed set of JVM threads.

Long-running actors can't monopolise or block a thread, otherwise the system may deadlock. This is achieved on the JVM by two techniques:

- Compile-time rewriting of the bytecode, to add the ability for a thread to pause an execution and give another one a turn.
- Runtime components for actors and threads that perform this distribution in an optimised manner.

The limitations of the JVM dictated that both of these techniques were needed to achieve a successful solution to the problem.

1.1 Contributions

The source code for the the Whiley language can be found at `http://whiley.org`, and the content of this project at `https://github.com/zimothy/Whiley/tree/actors`. Its contributions are:

1. An implementation of the actor model for the JVM using a combination of compile-time and runtime techniques.
2. The replacement of the existing concurrency system for the Whiley compiler while disturbing as little as possible of the existing software.

Chapter 2

Background

This chapter provides a necessary background for the project. It introduces the Whiley programming language, the Actor Model of concurrency, and code execution on the Java Virtual Machine.

2.1 The Whiley Programming Language

Whiley is a very young programming language [13]. At the time of writing the core language has only begun to settle down and the solutions to several problems in the language's specification have yet to be definitively determined. The syntax and core semantics are much more well defined though, and this is what is primarily needed for the remainder of the report.

The purpose of the language is to provide a modern and approachable base for safety-critical software [13, 14]. It includes a rather comprehensive constraint system which can verify pre- and post-conditions at compile-time. This allows a programmer to write more detailed specifications about the nature of a program than a typical type system allows, with the intention of reducing bugs in the resulting software. However, while an interesting facet of the language, the constraint system is not the focus of this report.

The language could be considered a hybrid of the Object-Oriented and Functional paradigms. Data structures are immutable, and the language supports side-effect free functions, combining the C style of function declaration with Python's indented code blocks. All variables are both statically and strongly typed, meaning the compiler knows precisely what possible types a variable's contents could have.

2.1.1 Data Structures

The top level of the Whiley grammar allows only type, function and method declarations. The syntax for type declarations involves defining an *alias* for a type:

```
define intlist as [int]
```

In this example, an alias `intlist` is created for the type `[int]`. Any appearance of this alias in the code is substituted for the original type.

The language has built-in syntax for lists, sets, tuples and records, and all of these can be used to build new types. Saving data in records is a common and convenient way to group named data together:

```
define point as { int x, int y }
```

This creates a type `point` for a record with fields `x` and `y`, both holding integers. An instance of this type can be created with the literal syntax `{ x: 1, y: 2 }`, which assigns the given values into the relevant fields.

The type of an unbounded list containing integers can be represented with `[int]`, and a set with `{int}`. The literal value `[1, 2, 3]` represents a list containing the three given elements in the same order, while `{1, 2, 3}` represents a set. A tuple type consists of a certain number of types in a specific order. For example, a pair consisting of an integer and a real number would have type of `(int, real)` and could be written as a literal `(1, 2.0)`.

2.1.2 Functions

Whiley's functions are distinct from methods in that they are always 'pure', that is, they cannot perform side-effects [16]. They take some input, and produce an output, without reading or writing to any state. This means they are *referentially transparent* — the same input will always yield the same output. The syntax for declaring a function is reminiscent of C, listing types before names, but the body is declared using indentation rather than braces.

```
int id(int value):
    return value
```

Functions are made up of a sequence of statements, and always return a value. Like Python, variables are declared by assignment, though unlike Python variable types are statically inferred to ensure type safety [14]. In fact, a variable's type can change over the course of the body.

```
(int, int) add(int i):
    j = i + 1
    j = (i, j)
    return j
```

In this code, variable `j` is assigned an integer, and then a tuple made up of integers. This is not only valid code but also well-typed: `j` is initially typed as an integer after the first assignment, but changes its type once the second assignment occurs.

2.1.3 Processes

Whiley begins to work more like an object-oriented language when it uses processes. Processes provide mutable state, and variables typed as processes are really references to this underlying data. A process type uses the `process` keyword in its definition.

```
define Person as process { name: string }
```

Prefixing a literal with the keyword `spawn` creates a new process value. The Erlang programming language uses this keyword to create new *actors* [2], and the same goes for Whiley. The nature of actors and the consequences of using `spawn` will be discussed in more detail shortly.

2.1.4 Messages

Processes can communicate with one another by passing *messages*. Passing a message is akin to invoking a method, except a process can only respond to one message at a time. If a process receives a message while it is in the midst of a response then the message goes into a 'mailbox', and the actor then processes this backlog in order of arrival. Messages are a core component of the actor model, and will be discussed in more detail in the next section.

Whiley has two different ways to send a message — *synchronously*, in which the sender waits for the receiver to finish responding, and *asynchronously*, where the sender continues their own work without waiting. The syntax for a message send involves listing the receiver of the message, the method the receiver should use to respond to the message, and the arguments to pass to the method. Which kind of message send is determined by the operator between the receiver and the method name.

Asynchronous sends use the `!` operator. This tends to be the preferred operation for causing another actor to perform an action if no value needs to be retrieved from the message.

```
person!say("hello")
```

In this case, the message `say` is sent to the `person` process, with a single argument `"hello"`. The sender does not wait for the receiver, but continues on to the next line.

Synchronous sends use the `.` operator. These are mostly useful for retrieving values from an actor, as blocking isn't usually the desired behaviour.

```
name = person.getName()
```

Here the message `getName` is sent to the `person` process, and the result of the value is saved. The sender must wait for the receiver, even if it does not intend to store the result of the message send.

It's important not to confuse the synchronous operator with the record access operator, as they both use the same symbol. If the right hand side of the operator does not include the argument parentheses, then it is a record access, not a message send.

```
sys.out.println(name)
```

In this example, the property `out` is first retrieved from the record `sys`, which is then sent the message `println` with a single argument, `name`.

2.1.5 Methods

Whiley's methods resemble its functions, except that they can interact with the outside world. Functions are for evaluating values whereas methods are usually for causing behaviour to occur [13]. The first kind are the so-called 'headless' methods, and syntactically only differ from functions in that they have their name prefixed by a `::` symbol.

```
void ::printName(System sys, string name):
    sys.out!println(name)
```

A method may also place a process type before this prefix. This attaches the method to the type and requires it to be invoked through a message send. These methods have access to a `this` variable, which holds the process the method is being invoked on.

```
void Person::printName(System sys):
    sys.out!println(this.name)
```

Synchronous message calls on `this` are not queued, but rather invoked immediately. Otherwise the process would immediately deadlock, waiting for itself forever.

This is entering into the concurrent nature of Whiley, so next we shall examine the relevant model of concurrency.

2.2 Concurrency and the Actor Model

Program concurrency is a problem that has been around for decades. Many programs require it in order to function correctly. Applications with a graphical interface tend to maintain the interface on a separate thread to prevent freezing the UI with other computation, and operating systems need to be able to run large numbers of separate programs simultaneously. The rise of the consumer-grade multicore chip has seen the need for parallelism take centre stage for those seeking performance gains in the last decade.

Despite claims to the contrary [5], there isn't a single solution which universally deals with the problem in the way that, for instance, garbage collection does for memory management. Heavyweight threading remains the common solution, but concurrent problems vary between domains and there a number of accepted solutions which excel in different circumstances. The *actor model* is one such solution, where program execution is segmented between *actors* who do not share any state [1, 7]. These actors communicate and synchronise by using *messages* [7]. These messages are the only form of communication the actors may have with one another, and so all the actors can run in parallel without illegal accesses between them.

There are several production-level languages which make use of the actor model. *Erlang* is a functional language which uses the model as its primary form of concurrency [2], while *Scala* is a multi-paradigm language which includes it alongside more traditional options like threading [8]. Libraries like *Kilim* provide the model to existing languages [8, 10].

2.2.1 Concurrency vs. Parallelism

The difference between concurrency and parallelism is important when it comes to discussing concurrent solutions. The important distinction is that a concurrent system allows multiple processes to run side by side but not necessarily at the same time, whereas parallel systems actually have these processes running simultaneously. A concurrent system may be parallel, in part or even in whole, but parallelism isn't a requirement of it. Instead, many concurrent systems merely weave back and forth between the processes, giving the appearance that they are running alongside one another.

The reason this difference is important is because exploiting both of these techniques is how the implementation for Whiley's actor model achieves its own concurrency. Java uses *threads* to achieve concurrency [11, ch. 8], and a naïve implementation could simply make every actor its own thread. However, in a lot of cases — this one included — such massive parallelism is not a viable option because JVM threads are too 'heavy' to exist in large quantities.

2.2.2 Actors

Actors separate the concurrent nature of a program into concurrent primitives whose logic is entirely contained with its own locality [1]. One actor cannot breach the locality of another, but they can communicate by sending messages to each other. What defines an actor's locality is dependent on the interpretation of the model. In Erlang, where state can only be found in the closures of functions, only arguments and local variables can be used by the actor. In Whiley each actor has its own mutable state which it can access at any point [13].

If several of the heavyweight threads attempt to access the same state at once then problems develop. For instance, if one thread attempts to iterate over a list while another modifies the contents, it's not clear what the outcome should be. The separation of state into actors allows each one to run concurrently without fear of causing these problems, because actors can only access the contents of their individiual localities. As all mutable state in Whiley is embodied in actors, it's not possible for two actors to modify the same underlying state. Other problems (concurrently reading and writing a file, for instance) are easier to avoid because the logic for that action can be contained and routed through a single actor.

It's important that an actor be lightweight, unlike the much heavier threads of the JVM. The expectation is that thousands if not millions of actors may exist in a system at a point in time if the design demands it. This is a possibility for modern object-oriented systems, but not for threads. As will be seen later, the penalty for attempting to use heavyweight threads in mass numbers is high. The clear implication is that if actors have mutable state then it is allowable for them to be Java objects but not Java threads, while also having concurrent behaviour. The purpose of the project was to solve this problem.

2.2.3 Messages

Although actors can't directly affect the state or behaviour of another actor, they can communicate with one another [1, 7]. Sending messages is a useful way to synchronise with the state of another actor, or generally just propagate behaviour to other parts of a system. In many actor models message sending is purely asynchronous [3, 8]: the sender carries on its execution regardless of the reaction of the receiver. Indeed, the receiver may not react for some time, as it may be caught up in other work. In these situations, an actor may perform a receive action, blocking until a new message arrives, and then responding with the appropriate action for the contents of the message. Synchronous behaviour can be built up through combining the send and receive actions.

Whiley extends this behaviour by removing the need for message management from the programmer [13]. An actor's mailbox holds all incoming messages in a queue and then processes them one by one in the order in which they arrived. Actors have no way of explicitly waiting for a message to be received, but rather may send a synchronous message to another actor. This will cause them to wait (or *block*) until the receiver responds, allowing looser coupling between the behaviours of actors.

Messages aren't just for triggering behaviour. The internal state of an actor can be retrieved and assigned to using synchronous messages if the receiver provides methods for it. This way information can be passed around the system in an object-oriented style while the synchronisation of the concurrent components is handled implicitly. Choosing what kind of message send is most appropriate is all that is needed beyond this.

When a Mailbox empties and an actor has no more messages to process, it is considered *idle*. Idle actors may still be relevant to the system, as there is ample opportunity for them to be sent more messages. An idle actor should not necessarily be destroyed, but it is important that they do not attempt to claim resources like processing power if they have no intention of using it.

2.2.4 Spawning

Both Erlang and Whiley make use of the `spawn` keyword to create new actors [13, 2] but they work in very different ways. In Erlang, `spawn` is followed by a function call, which will execute asynchronously from the function which called it [3]. Thus is a new actor created and a unique identification returned which can be used to send messages to the actor. It's important to note that the newly spawned actor does not have access to the ID of its creator unless it receives it in a message. In this way the actors are separated into a hierarchy of parents and children.

Whiley uses this hierarchy as well, and `spawn` also creates a new actor and returns a reference to it [13]. But the crucial difference is that beyond this, no other action is taken. In Erlang the new actor immediately runs the function call that it was spawned with, but in Whiley every actor is born idle. Spawning an actor is about creating the process and its mutable state, not asynchronously calling a function.

It's not hard to see why this difference arises. Erlang is a functional language with no mutable state. Their actors can't go idle, because they're not chunks of data, they're a running function. An actor is either currently working or waiting for a message to be received. In Whiley's case, spawning isn't about creating new behaviour that is diversified by messages, it's about *creating new data over which behaviour can be triggered*.

2.3 Java Virtual Machine

The purpose of a virtual machine is to implement the same platform across a number of environments, allowing code that executes on that platform to be run on any of the implementations. The programming language *Java* has a runtime environment that is particularly prolific, with implementations of the Java Virtual Machine (JVM) for a wide variety of platforms. From desktops to mobile phones, a JVM can usually be found. This is why so many languages other than Java target the JVM interface — Clojure and Scala are examples of languages designed explicitly for the JVM [6, 8], while Python, Erlang and Ruby all have ports as well [15, 20, 18]. Whiley falls into the former category [13].

2.3.1 Stacks and Bytecode

The purpose of Java bytecode is to provide an assembly-like language which the JVM can more easily analyse and verify before execution. Within a method the code has access to a stack and local variables [11, ch. 7] and, like assembly, it uses unstructured control flow with labels and variations on 'goto' statements to traverse the code. Native OO concepts like object creation and virtual method invocation are built into the system.

The JVM maintains a *call stack* that records what methods the execution is currently in, with each one represented by a *frame*. The frame on the top of the stack is for the current method. If it invokes a method then a new frame will be pushed onto the top of the stack, and when it returns it will be popped off (the frame underneath will once again be executing) [11, ch. 3]. While the local variables of each method actually all exist in the same stack, the compiler will verify that the code does not attempt to access values that exist outside of its frame [11, ch. 7] (so there is no harm in visualising each of the local stacks as different).

The bytecode must be well-typed in order to pass the JVM's verification. This means that the compiler must be able to compute exactly what kind of values will be on the stack and in the local variables at any point in the code, before or after any bytecode operation. The nature of the stack consists of its height (how many items will be present at that point), and the type of each individual item held. The nature of local variables describes which are in use, and what the type of the variable is. Unlike Java, the type of a local variable may change, so long as use of the type is consistent in the code. Consider this Java snippet:

```java
void run()       { id(5); }
int  id(int x) { return x; }
```

If the thread enters this program at the `run` method, then a frame for that method appears on the call stack. It pushes `5` onto its local stack and invokes `test`, pushing a new frame onto the call stack. This new frame will have `5` in its first local variable, as this is the value passed for the first argument. The frame is then popped off the call stack, and the frame underneath returns to where it was before and continues on.

Chapter 3

Design

The chief question behind the development of any new programming language must be *why?* Why a new language when there are already hundreds out there? Is it because a problem would be better solved with its own domain-specific language? Or because no other language quite captures the broad philosophy that is now needed? Why should a programmer choose this new language, and what solutions does it offer them?

Easing the pain of concurrent programming could be part of an effective answer to this question. Many modern languages claim concurrency as one of their paradigms, and although it would be facetious to claim that C or Java aren't concurrent it's fair to say that they don't make it easy.

Making the design and implementation of concurrent programs more simple is one of the problems that the Whiley programming language attempts to solve. Whiley has adopted the Actor Model, a segmentation of processes into concurrent primitives, which neatly matches the structure of the language's Object system.

The problem that this project has attempted to solve then, is this: that Whiley uses the Actor Model as its model of concurrency, but it runs on the Java Virtual Machine which has no built-in support for such a model. How can we implement Whiley efficiently on the JVM?

3.1 Motivation

It's not hard to see why achieving concurrency is a necessary aspect for many programs. Some examples have already been given, and others aren't hard to come by: a server with a high load would benefit from being able to deal with multiple requests simultaneously rather than in sequence, whilst a divide-and-conquer algorithm could compute solutions in parallel. The question isn't *why* we would want concurrent behaviour, but *how* we go about achieving it. If threads give us the parallelism asked for, isn't the problem solved?

The answer is, of course, that it isn't that simple. Threads run independently with little limitations on their interactions with data, and where multiple threads interact problems tend to coalesce. The problem is *synchronisation* — any interaction between threads needs to be both carefully considered and manually implemented. Most importantly, the compiler will not prevent these issues from arising.

Concurrency introduces a host of problems with *shared state*. For instance, incrementing a variable translates down to three atomic actions: retrieve the value out of the variable, change the value, then place the new value back into the variable. Because this isn't just one atomic action, there are opportunities for another thread to modify the variable between reading and writing it, meaning we haven't incremented it properly. This is called a *race* *condition*, and is a common problem in concurrent programming.

The JVM's threads are also very *heavyweight*. This is a common informal description which means that they incur a heavy performance penalty when created, as well as a large memory consumption throughout their lifetime. Similarily, use of the term *lightweight* refers to *not having either of these properties*. What 'heavy' and 'large' mean will vary between different systems, but for the most part platforms will not permit thread creation on a large scale. Even moderate use of threads may incur greater costs than simply not using them at all.

Because of this restriction, it's more difficult to structure a concurrent program. A programmer can't create a thread for every concurrent structure, and it often isn't clear what the optimal number of threads should be. It's often safer just to use threads as necessary despite the performance gains that parallelism offers because of the problems they tend to introduce.

The actor model solves both of these problems. Race conditions are avoided because such actions are contained within the locality of an actor, so another thread's execution can't intrude on it. The messaging system allows other threads to request the change, but they can't make it directly, and only one thread can be executing the behaviour of an individual actor at one time. Moreover, actors are much more lightweight than threads, meaning a programmer may design the system around as many concurrent processes as is desirable without worrying about the thread count. Achieving an optimal thread count is handled by the backend of the system, something a programmer need never touch.

This isn't to say that actors are a universal panacea for every concurrent woe. The programmer is still responsible for the design of a program and race conditions can still arise at a much higher level. The point is that the model allows stronger reasoning about the relationships of software's concurrent components, avoiding common problems while achieving the goal of parallelism.

3.1.1 The Problem

The core of the problem is that the JVM is not inherently designed for such a concurrent model [11, ch. 8]. Actors must be lightweight and concurrent at the same time, but the only concurrent component of the JVM is the heavy thread. It's possible to build concurrency into a sequential system by weaving back and forth between executions. The JVM is not well designed for this feature either — halting a method's execution is possible with a return statement, but how do we resume it later on? The design of the solution needed to take these factors into account, and attempt to combine this lighter method of concurrency with the heavier threading.

3.2 Solution Concepts

In order to implement the actor model on the JVM, a number of techniques are required. Halting the execution of an actor in order to let another have a turn can be achieved with a *continuation*, where the local state of a method is bundled up into a single value in order to allow the execution to continue at some point in the future. Building a continuation involves a technique called *yielding* which details the halting process, and once constructed it can be handled by a *scheduler* which will manage when to effect a resumption. This section will discuss these three techniques.

3.2.1 Continuations

A continuation represents the *control state* of a frame on the call stack at a given time [17]. What this control state consists of depends on the nature of the program, but typically includes where in the code the execution is currently at and the values of the local variables in the frame. When a continuation is held as a value it corresponds to a paused actor elsewhere in the program [19], and allows the resumption of this actor because it has stored the control state. It merely needs to restore this state in order for the actor to resume executing, returning to the way it was before being paused.

Lightweight concurrency is possible through shifting which actor has control of a thread before it has finished executing. In order to achieve this the actors need to be paused and resumed at specific points. Continuations are created at the point a actor pauses, and used when it resumes. The creation of a continuation needs a definition for the control state of the relevant platform. To illustrate this point, consider what the control state would be in a Whiley method.

```
int ::method(int x):
  y = x + 1
  // What would the control state be here?
  return y
```

If a continuation were to be formed when the method's execution is at the point where the comment appears in the code above, its control state would consist of two parts. The first is the location of the execution. If the method were to be resumed in the future, clearly it should not start again at the beginning. The second is the state of the local variables. The value of the variable `y` certainly needs to be restored in order for the return value to be as expected, but a continuation could choose not to save the value of `x` as it will not be needed after the resumption.

3.2.2 Yielding

A continuation stores the control state to allow for pausing to be reversed, but yielding actually achieves the pause [9]. It draws its name from the idea that the concurrent actor 'yields' their execution to allow for others to have their share of thread time. Figure 3.1 illustrates an example of how this helps in comparison to a purely sequential format. Without yielding if one of the actors never finishes then those waiting for access to the thread are never executed.

This high-level concept of yielding is quite simple, but the actual procedure for a platform like the JVM which uses a call stack is much more complicated. The currently running method may have been invoked by another method, so if the frame is to yield then all of the frames underneath it in the call stack must yield as well. This process is called *unwinding* the stack.

Although yielding allows for lightweight concurrency, the actual process can be expensive. Sometimes it is more efficient to let an actor finish its operations to free up its thread rather than pause it, so choosing when to yield is an important part of the process. A system which intends to insert yielding automatically needs to have a good understanding of when it will benefit the program and when it will not.

3.2.3 Scheduler

When the call stack is unwound the continuation that is generated must be used to cause a resumption in the future, when the thread is free again. This design calls for a scheduler

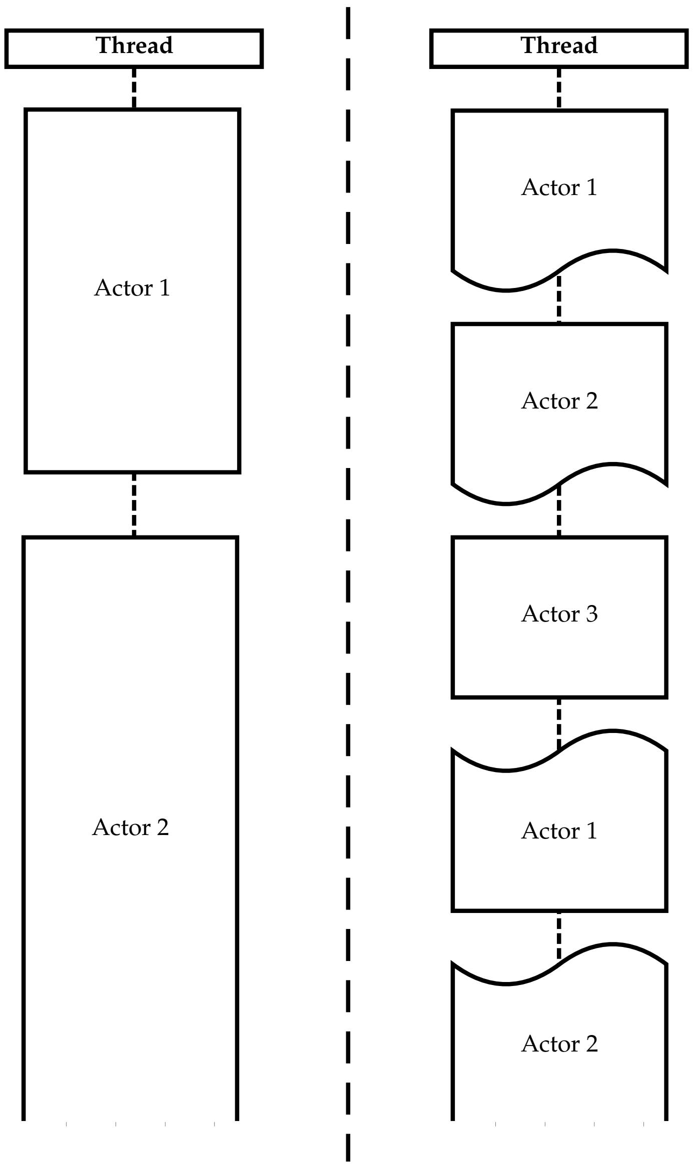

Figure 3.1: Two ways in which three actors can share one thread: with and without yielding. In the sequential version on the left, actors run to completion before stopping. This means that other actors cannot gain access to the thread until the complete. Actor 2 never completes, so in the sequential version Actor 3 never executes. Yielding allows an endless actor to free up room for others to run.

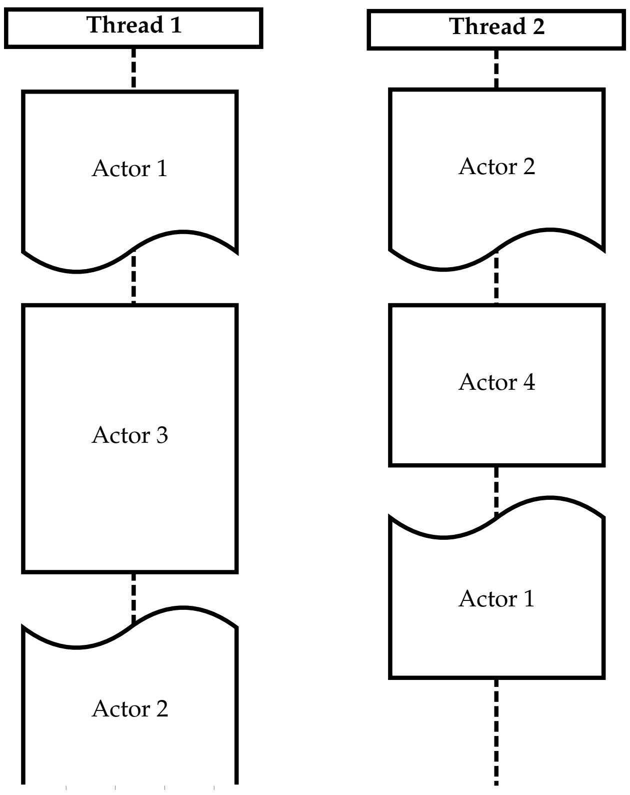

Figure 3.2: An example of four actors sharing two threads.
When Actor 1 and 2 resume, they have swapped threads.
Which thread an actor runs on doesn't affect its execution.

that is responsible for deciding when an actor yields and which should be resumed next. The order that they resume affects efficiency, but simply scheduling them in the order they arrive is acceptable. In order to make these decisions, the scheduler needs to be aware of whether or not the thread is free, as well as which of the actors waiting on a yield should be given preference.

Because the scheduler can make these decisions, the concurrent actors can be distributed among many different threads. When an actor resumes it need not return to the same thread it was running on before — so long as the continuation restores the control state, it will resume correctly. Whereas figure 3.1 illustrated actors sharing one thread, figure 3.2 shows how it might take place when multiple threads are available.

3.3 Solution Specifics

Given the concepts needed for implementing the lightweight threading, the design next needs to consider how the solution will integrate with the existing Whiley codebase. This involves attaching new components to the Whiley compiler as well as designing how the concepts will translate onto the JVM.

3.3.1 Integrating with the Compiler

The Whiley compiler is an ongoing piece of work, so the design needed to integrate the project as non-intrusively as possible to avoid as many dependencies as possible between the two. The new components are split into two sections: the *runtime components* and the *compile-time components*. These were integrated in different ways.

The runtime components are the actors and the scheduler. As the existing compiler already provided an Actor class, the new components needed only replace the behaviour of this existing class behind the same interface. This allows the new actors to be used seamlessly in all parts of the existing system despite the change in behaviour. Section 4.2.2 details how the scheduler fits into this design.

The existing compiler also provides a mechanism for easily integrating compile-time components. It uses Visitor transforms [4] in a pipeline — each transform registers itself with the compiler, and when compilation occurs the content is passed between the transforms in the order they registered.

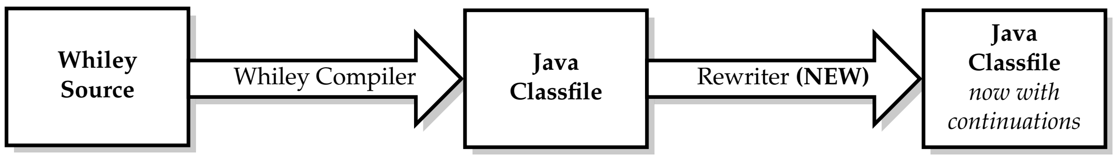

Figure 3.3: The bytecode rewriter's role in the compiler pipeline. The Rewriter component was developed as part of this project.

Figure 3.3 shows how the compile-time component, which will rewrite the code to include continuations, can easily be attached to the end of the pipeline. As a transform it will receive what *was* the final output of the compiler and make necessary changes to support continuations. This will be discussed further in section 4.1.1.

3.3.2 Messaging Logic

The semantics for messaging was discussed in section 2.2.3. The two different kinds of message sends cause different effects on the sender, and choosing when to yield control of a thread relates very strongly to when messages are sent. The order that messages are received in also needs to be recorded.

Figure 3.4 illustrates an example of messaging. Note that when synchronous messages are sent the sender is blocked, and only resumes again once the response is received. After an asynchronous send the sender is still executing, though. The actor *may* yield at this point, but it is not necessary within the semantics of the actor model.

The other feature of the diagram is the order that Actor 3 performs the messages in. In this case it is simply moving to the only message left in the mailbox, but if another message arrives after Message 3 was sent but before Message 2 completed, Message 3 **must** still be the next one processed. This means the mailbox must act with the 'First In, First Out' policy.

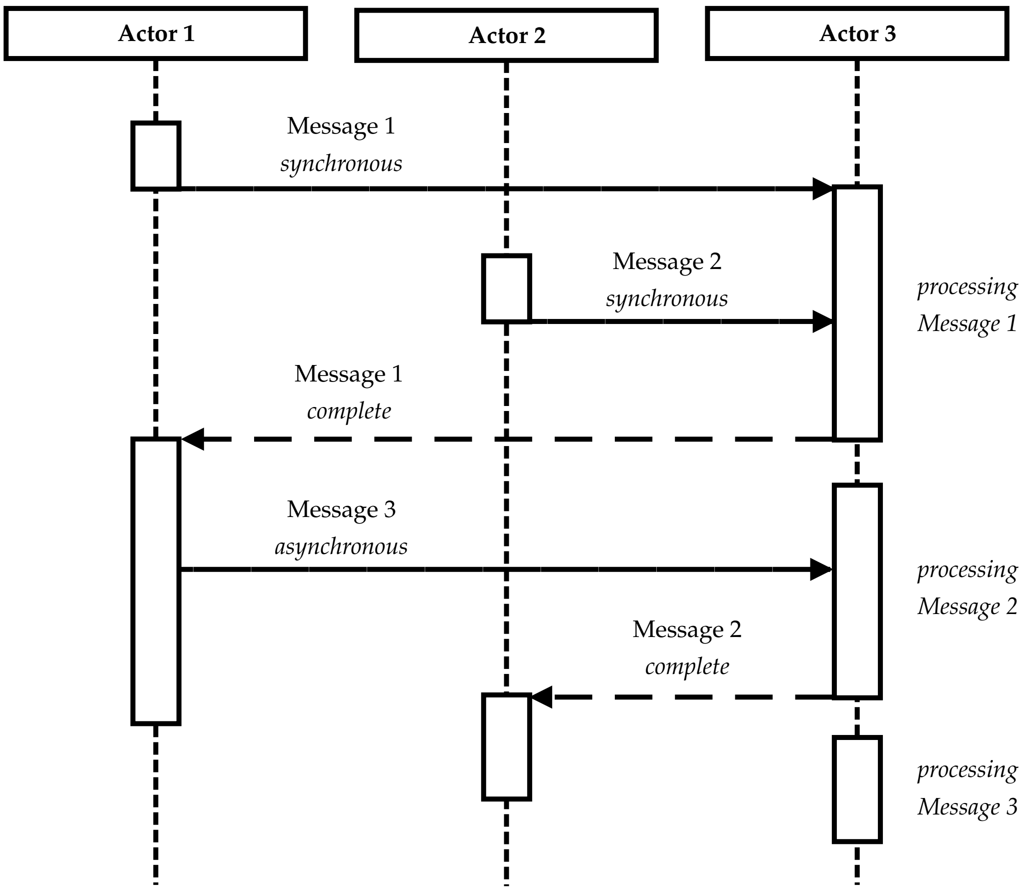

Figure 3.4: An example of three actors messaging one another.
Actor 3 will not begin processing Message 2 until it has finished Message 1.
Also, since Message 1 is synchronous Actor 1 is blocked until the message is completed.
Message 3 is asynchronous however, so Actor 1 is free to continue after sending it.

Chapter 4

Implementation

Perhaps the most important aspect of this implementation is that the Whiley compiler is already a burgeoning piece of software. The new solution needed to integrate into that software while disturbing as little as possible of the existing design. However, this helped prevent unnecessary coupling between other elements of the compiler.

The software design is split between two separate entities. The first is the compile-time logic, which modifies the JVM bytecode of the Whiley methods in order to package up the continuations and cause yielding to occur. The second is the set of runtime components that will exist as objects while the Whiley program is running. These contain all of the concurrent logic including the distribution of actors over threads, the messaging system, and the runtime continuations.

Dealing with concurrent behaviour tends to lead to unexpected behaviour. Identifying and solving these problems provides a more effective understanding of the nature of the system and how different concurrent components can interact with one another. There is one problem in particular that this chapter will discuss.

4.1 Compile-Time Components

The continuations themselves are runtime components because they are values that exist at runtime, but choosing where in the execution to make them and what to put in them is something that must be done at compile-time. The JVM doesn't support the idea of yielding and saving control state, so it needs to be added manually into the classfile's bytecode by the compiler.

Section 3.3.1 explained how additional compile-time components can be integrated into the Whiley compiler as a code rewriter. This rewriting will pick points in the code to yield at. Choosing effective yield points automatically is a difficult problem and outside the scope of this project. For now, they are placed after all message sends.

At any of these points, the rewriter must also add bytecode to save the control state into the continuation that will be generated. Because the bytecode is well typed, the rewriter can know at compile-time what the size of the local stack will be and which of the local variables are in use. It's not possible to loop over the contents of the stack or the variables, so it will need to use this information to generate the continuation.

4.1.1 Continuations

The Continuations class is at the top-level of the compile-time design, a transform which will perform the appropriate rewriting on a given class. For this purpose, it has a method

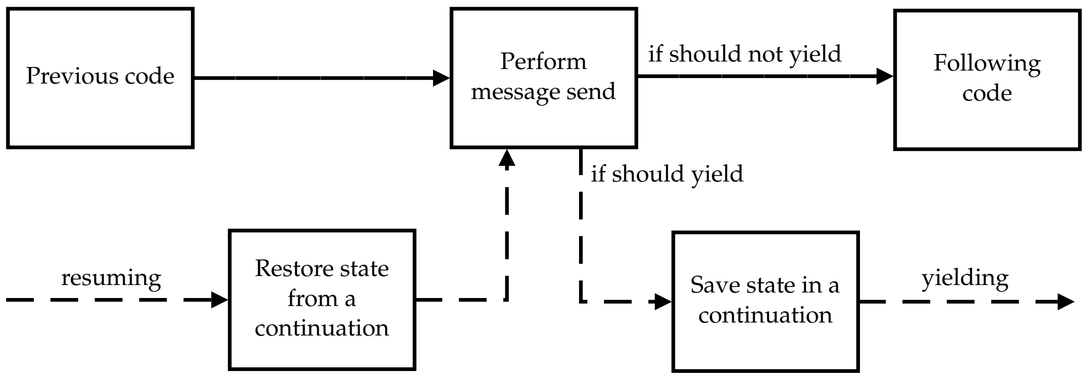

Figure 4.1: The execution path of a message send.
The dashed arrows mark the flow added by the rewriter.

which takes a classfile. This method hides the much more detailed structure that is necessary for the rewriting algorithm which is shown in figure 4.2.

For each method in the given class the algorithm extracts the bytecode and performs the rewriting algorithm on it. This algorithm looks for message passes to an actor (see section 3.2.2). At these points it places requests for whether the execution should yield, as well as yield and resume points after and before the invocation respectively. All of this logic is illustrated in figure 4.1, showing which parts existed before and which have been added.

4.1.2 Packaging the Data

The difficulty in adding this new behaviour to the JVM is that there is no simple process for saving the stack and local variables. Because the bytecode is well-typed, however, it is possible to calculate the stack's size and which local variables are in use at any point in the bytecode. This is achieved bya *type flow analysis*.

Each different bytecode operation has a defined effect on the current control state. The bytecode operation `iconst_1`, for instance, always results in an integer added to the stack, while the `goto` operation jumps the current execution to a different point in the code without changing the types. The algorithm uses this information while running through the operations to build up exactly what the types must be at any particular point.

The higher-level aspects of the two analyses are exactly the same. As stated in the example above `goto` does not modify the actual type data, but it does copy the assertions about that data to a new location. This kind of behaviour can be handled by a single algorithm while the specifics of each analysis are branched off. The example states that `iconst_1` modifies the stack, so this would be part of the stack type analysis. The analysis for the local variables can simply ignore that operation and move onto the next one without changing its type data.

Because the analysis needed for the two different states is very similar, most of the behaviour is packaged up into the TypeFlowAnalysis class. It uses a template method [4] which provides the actual algorithm while its two subclasses just provide specifics about what information to gather. These subclasses are StackAnalysis and VariableAnalysis, and can be seen in figure 4.2.

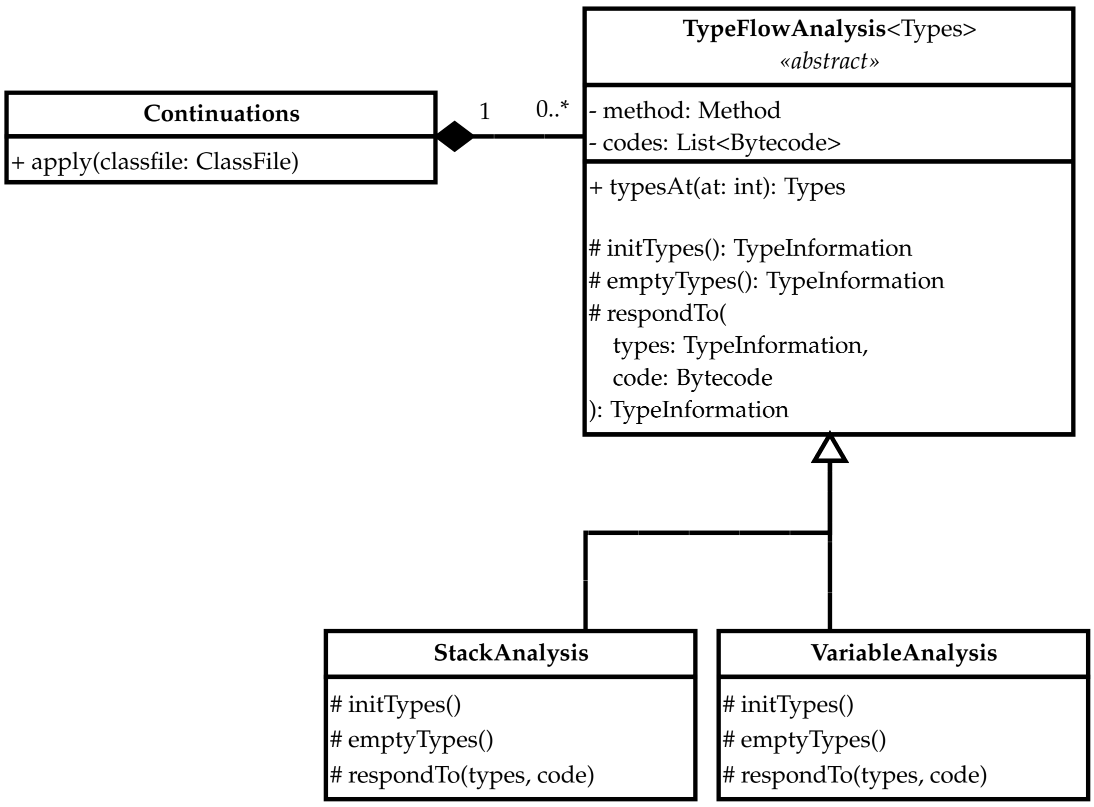

Figure 4.2: The class design for the compile-time components

4.1.3 Saving and Restoring

Once the rewriter has the relevant type information it is trivial to save and restore the data in a continuation. For the local stack, the method that saves or restores the value from the continuation is invoked which respectively places or retrieves the value from the stack. For the local variables the same techinique is followed, but the value needs to be moved onto the stack before being saved and moved off it after being restored.

After saving the data the method then returns as part of the yielding process. Those frames which are underneath it on the call stack will then follow the same process. Restoring merely approaches this from the other direction. Section 4.2.1 discusses how the actor actually stores the continuation internally to account for the structure of the call stack.

4.2 Runtime Components

The existing compiler already provides an Actor class, so the new behaviour can be added on top of it. This avoids the need to dictate the interaction between these new components and the rest of the system. In order to maintain separation of concerns the three kinds of runtime behaviour — yielding, messaging, and statefulness — are separated out into three different classes. Actors inherit from these to gain all of the behaviour at once.

Each behaviour is split into an abstract superclass which exists solely to be subclassed by Actor. The yielding and continuation behaviour exists in the Yielder class, including the logic for saving and restoring the control state of a thread. The Messager class handles the communication of actors, performing message processing and synchronisation.

Individual actors aren't responsible for the threading anymore, as they need to be distributed across a common thread pool rather than attached to specific threads. Alongside the actor hierarchy is a Scheduler class which contains the threads and the behaviour for 'scheduling' an actor's entrance to them. Figure 4.3 shows the actor hierarchy and the scheduler's relationship in it.

4.2.1 Yielder

The design for the Yielder class takes a number of required behaviours into account. Figure 4.4 gives a more detailed insight into the nature of the class. When the bytecode requests whether the actor is yielded, this is where it sends the request to. Note the two different ways to save state in this class: get and set for the local variables, and push and pop for the local stack.

Also note that the State class that has emerged. As expressed in figure 4.3, the 'state' relationship between the two classes is a stack. This is closely related to the call stack — each State object is the saved control state of an individual frame.

This design flips the stored state of the call stack upside down, as seen in figure 4.5. At each frame on the call stack, the data is packaged up into a State object and then stored in the actor. As the call stack empties the other fills up with these State objects. This reordering is helpful for the resumption because it must start by restoring the control state of the call stack's *bottom* frame, which is the topmost State. Because of this structure, the resumption process simply follows the yielding process in reverse.

4.2.2 Messager and Scheduler

The Message class adds the ability to pass messages between actors, as well as coordination with the Scheduler class. Figure 4.6 details their relationships. Most of the runtime

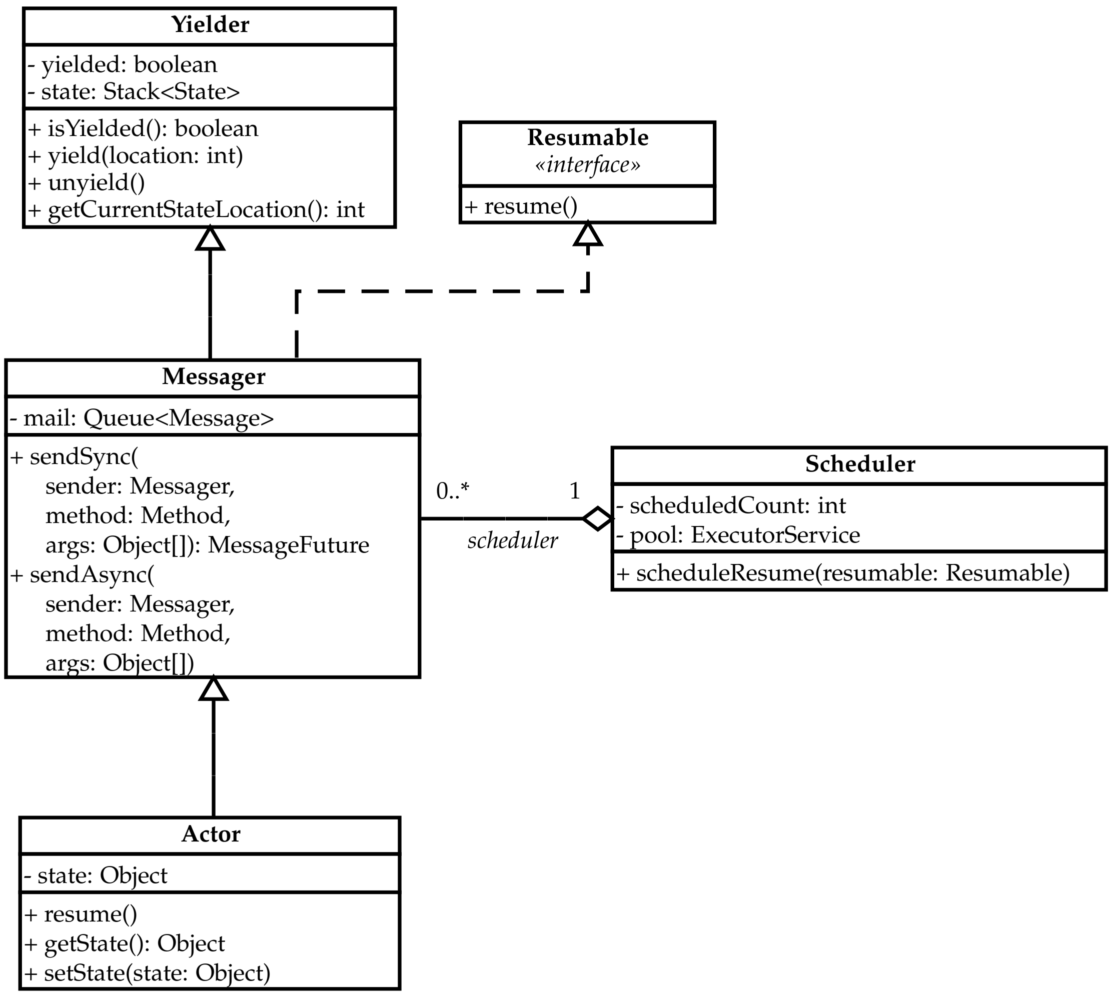

Figure 4.3: An outline of the design for the runtime components

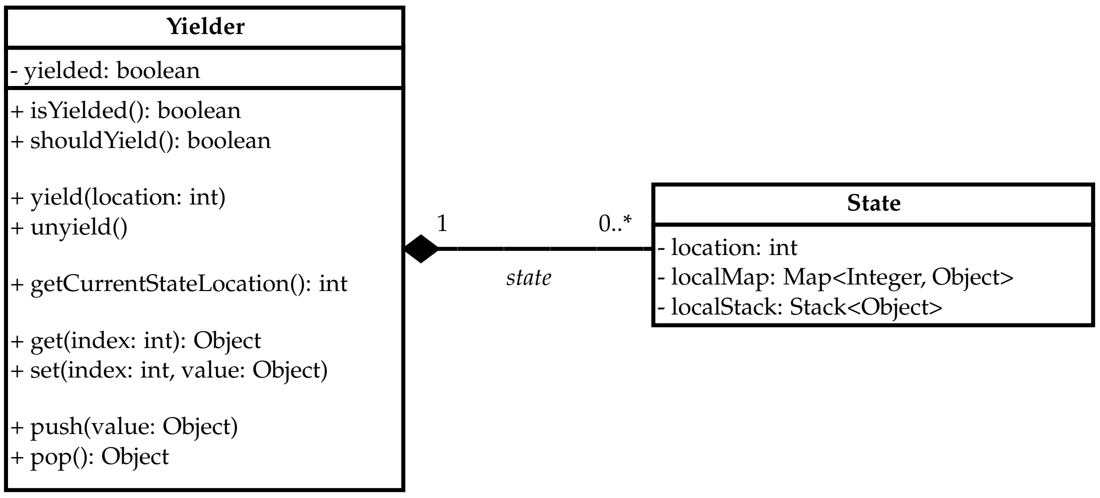

Figure 4.4: The detailed design for the Yielder class

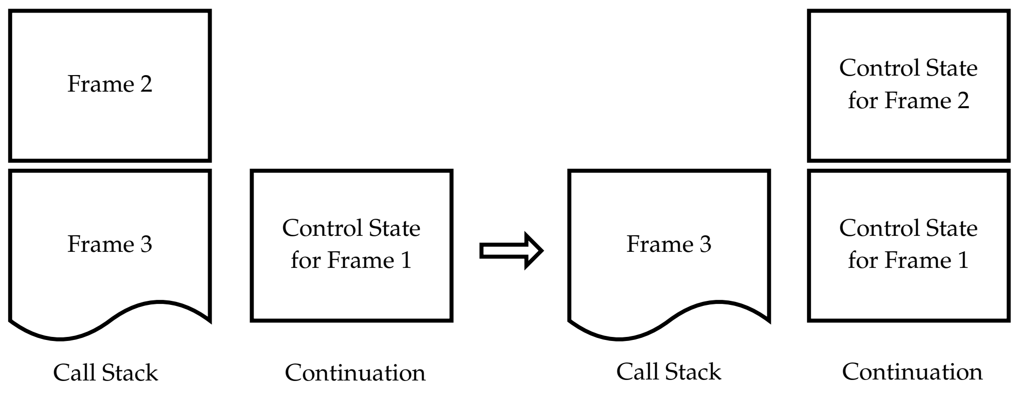
Frame 2
Frame 3
Call Stack
Control State for Frame 1
Continuation
Frame 3
Call Stack
Control State for Frame 2
Control State for Frame 1
Continuation

Figure 4.5: The relationship between the call stack and its continuation.
As the yielding progresses, the call stack empties while the other fills.
The control state for Frame 1, the top of the call stack, is the bottom of the continuation.

logic is contained within these classes, including all of the concurrency. Messager keeps its messages in a queue, with the topmost element being the message it is currently responding to. It exposes two methods, which differentiate only on the kind of message to send: synchronous or asynchronous.

Using the synchronous send creates a special SyncMessage object which has the added properties of remembering its sender and holding on to a *future* — a value to be returned immediately and which does not hold the expected result of the message, but that expects to hold it at some point in the future. When the response completes the sender can be alerted and the future filled with either the result or the cause of its failure. The sender must ensure the future has not failed before retrieving the result.

The Scheduler class is responsible for distributing actors across threads. It achieves this with a *thread pool* which holds a static number of threads and allows for asynchronous scheduling. When the scheduler requests that the pool execute a message it *will not block* regardless of whether a thread is available to execute it. This is important because otherwise a sender would block its thread while waiting for the other to resume, likely resulting in deadlock.

If a thread is not available, the actor is placed in a queue until one is. This also dictates the ordering: the actor at the head of the queue will be the next to be executed when a thread is free (Message priority may be a future concern). When all of the threads are idle and the queue is empty then there cannot be anything else to execute, so the scheduler shuts down and allows the JVM to exit.

4.2.3 Actor

The Actor class is significantly less complicated than the others, which was the reason for the separation of concerns. As seen by the preliminary design, it holds an internal state which is gettable and settable. When constructed it is capable of retrieving and using the scheduler for the actor which created it, simplifying the creation process.

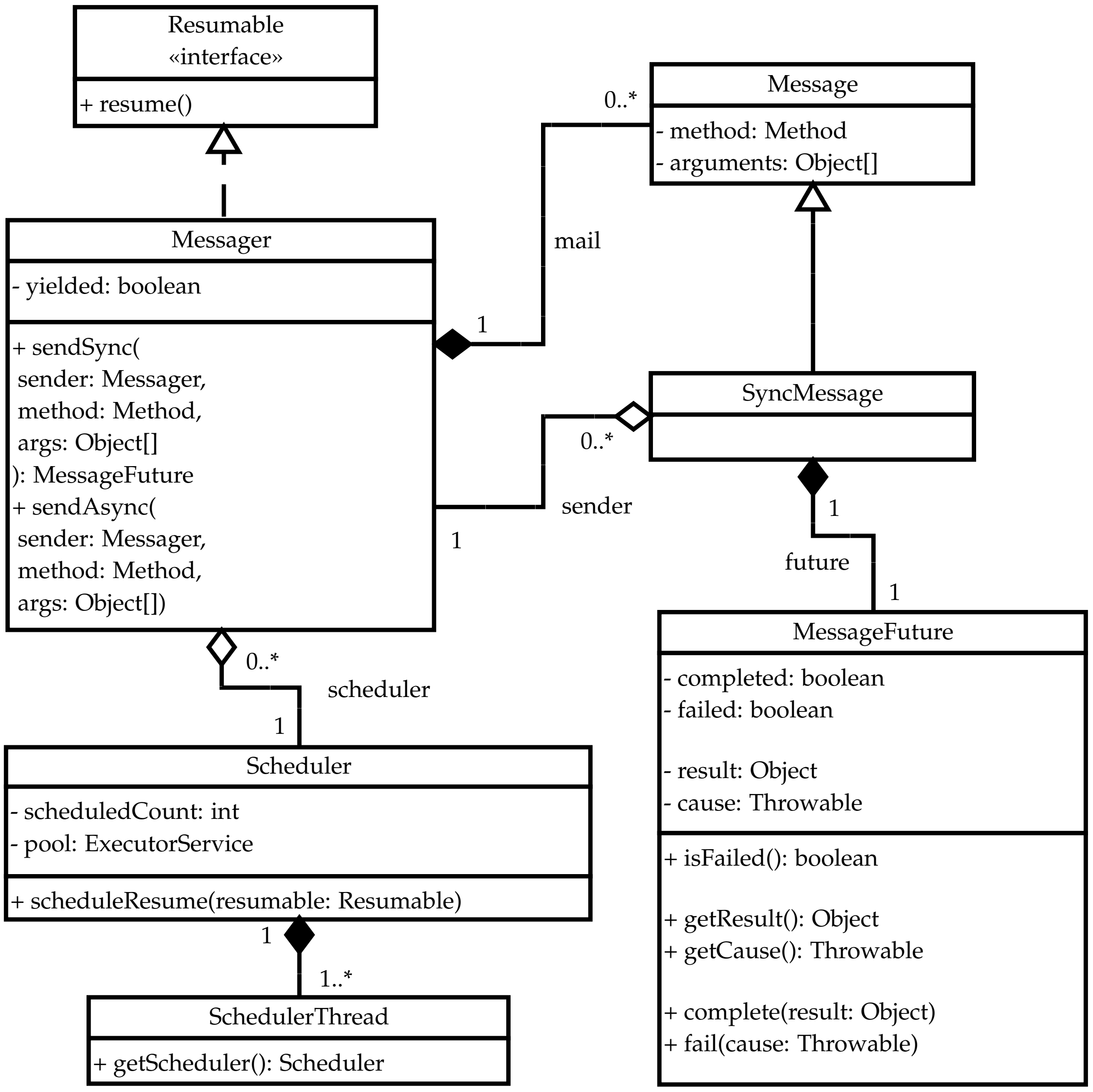

Figure 4.6: The detailed design for the Messager and Scheduler classes

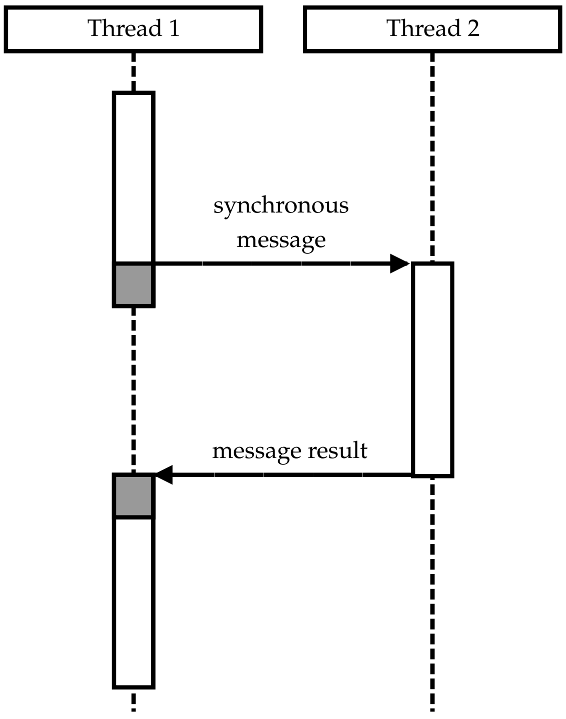

Figure 4.7: The normal behaviour of a synchronous message send.
The grey areas on the figure represent the yielding and resuming of an actor.

4.3 Race Condition

In order to achieve the best performance from the runtime components it is important to perform as little *synchronisation* as possible. Java allows a thread to declare that it must have sole access to data to avoid concurrent modifications. Using this technique is important, but it is also important to not be overzealous with it. The more it is used, the less parallel the active threads can be.

This section discusses a particular problem encountered during the implementation that could not be solved with Java's built-in synchronisation because it spanned too great a divide, through most of which actual synchronisation was not required. The problem was a *race condition* — sometimes an actor completed a message more quickly than expected and the sender was not ready.

4.3.1 Caught Yielding

This problem was *nondeterministic*, which means that it did not always happen, even if the input was exactly the same. Thread synchronisation is notorious for causing these sorts of problems because threads can run at different speeds each time a program is run. In this case, the problem was that the actor which had sent a message had not finished yielding control of its thread before it was resumed.

Figure 4.7 illustrates the normal behaviour for a synchronous message send, this time including the point where yielding and resuming occur. Figure 4.8 shows how the race condition could occur. Because the actors are distributed across threads, nothing stops them appearing on more than one thread at a time.

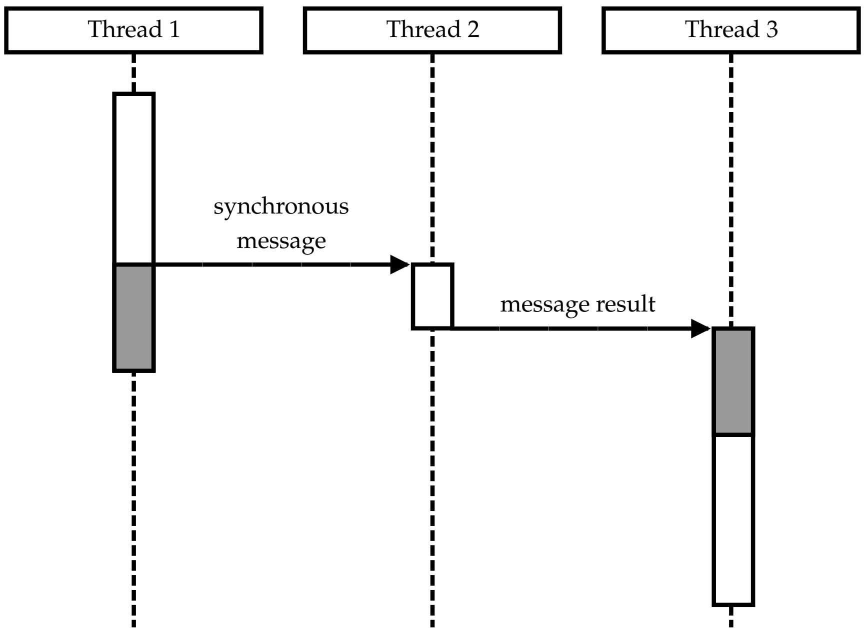

Figure 4.8: The erroneous behaviour of a synchronous message send.
The actor on Thread 3 is actually the same as the one on Thread 1.
This clearly violates the principles of the actor model.

If the receiver is too quick and the yielding too long then the receiver can signal a resume to the sender before it has finished yielding. The sender's thread is still occupied, so instead it is added to an idle thread where it runs the resumption in parallel with the yield. The continuation is not prepared for this sort of behaviour and so the system crashes.

4.3.2 Not Yet Ready

This race condition is an interesting insight into the complex nature of the solution's execution path. The Messager class is the only place where synchronised code is placed (apart from an unrelated occurrence in Scheduler) because the actor model strictly defines where concurrent communication can and can't occur. The Yielder class, which deals with the continuations, shouldn't need to make use of synchronised code because there shouldn't be any way for two threads to interact with it.

The solution to this particular problem isn't overly complicated. The Messager class includes logic for whether or not the yield operation has completed: if a resumption alert arrives when the actor is not ready, it does not attempt to resume at that point. Instead, it remembers that the alert arrives and promptly resumes once it is ready to do so.

Implementing the solution was trivial, but deciding where to place the relevant code took some time because of the concurrent nature of the program — even identifying the cause of the behaviour was a difficult task. Understanding where the execution was going inside the actor, and how another actor communicating with it affected that, is really important to finding and removing these problems.

Chapter 5

Evaluation

There are two stages to the evaluation of this project's outcome. The first is establishing the correctness of the solution: whether the code operates within the semantics of the actor model without the number of actors unduly affecting the stability of the program. The second is a measure of performance, comparing the speed of this solution to the existing Whiley platform.

This chapter aims to answer these questions through rigorous testing of the solution. For testing purposes both unit tests and benchmarks were run on a variety of systems with different specifications and operating systems. All systems were running version 1.6.0 of the JVM when these test results were compiled.

Target Platform	**Operating System**	**Processor**	**Cores/Threads**	**RAM**
Low-end consumer	Mac OS X Lion	1.4 GHz Intel Core 2 Duo	2 / 2	2 GB
High-end consumer	Windows 7	3.3 GHz Intel Core i5-2500	4 / 8	8 GB
Remote server	Ubuntu Linux	2.4 GHz Intel Xeon e5620	4 / 8	512 MB

Table 5.1: The specifications of the three systems used for the evaluation

5.1 The Solution

In determining the success of the project the first question must be whether the solution solves the problem. Section 3.1.1 defined the problem as not only adding the actor model to Whiley but also in a way that uses lightweight threading to avoid the costs incurred by large numbers of Java threads. In order to establish the solution's success, it must be shown to work as expected *and* deal with large quantities of actors.

5.1.1 Correctness

A small number of unit tests were created to verify the correctness of the implementation. This isn't very satisfactory for measuring its success, but the number of situations that need to be covered by such a suite is simply too large to have expected such an output from this project. Whiley already has a very large test suite of its own that should cover a lot of ground, but it is expected that more tests will be made in the future.

Simple tests cases turned out to be quite useful as well. The race condition discussed in section 4.3.1 only arose as the result of a small number of actors being active at once. These are the sorts of edge cases which will be difficult to comprehensively test for, considering their nondeterministic nature. The benchmarks used for performance evaluation served as examples of larger, more practical programs.

5.1.2 Lightweight Threading

In order to supplant the existing solution, this new solution must scale up to massive numbers of actors. Obviously there is a physical constraint on the number objects that can exist, but memory constraints are much lower for threads than they are for lighter objects. The existing solution maps each actor to its own Java thread. This allows the easy comparison between the heavyweight and lightweight threading styles by simply increasing the number of actors and observing when they crash from running out of memory.

This test uses the square of the input size in actors to solve a problem, to let it scale up the number of actors quickly. A input of 10 would use 100 actors to solve the problem, for instance. For the purpose of the test the input size was increased until the test failed to determine the exact point of failure. The solution failed the test if the JVM threw an exception about the memory constraints of the execution, and passed if it reached the input size of 1000 (one million actors) without exiting prematurely.

For the existing solution, the Linux server capped out when it attempted to solve for an input size of only 26. The JVM would not allow 676 threads to be active simultaneously and exited before solving the problem. This illustrates the problem perfectly — 676 actors is not an acceptable limit for the model. The laptop, which has twice the amount of memory, failed the test for an input of 51 (2601 actors), which is also disastrously low. The test never officially failed on the desktop (which has a significantly superior amount of memory) but as it approached 10,000 actors the whole operating system began to slow down substantially and eventually became unusable.

The new solution successfully solved the problem up to an input size of 1000 on all of the systems. This means the upper bound of actors is unestablished. The lightweight actors are much smaller in memory cost than threads but it was considered outside the scope of this evaluation to determine precisely how much smaller.

5.2 Performance comparison with Whiley

Having established the superiority of the new solution over the existing one regarding input size, the remaining comparison is how well each performs. Because threads have large initialisation overheads as well as memory costs it was the expectation that the existing solution would slow down considerable for larger actor counts. However, the parallelism achieved by smaller numbers of threads before the overheads become apparent might prove to be successful. It was determined that it would be appropriate to test performance at both large and small actor counts.

5.2.1 Benchmarks

This testing was achieved by the use of several benchmarks written in Whiley which used actors on varying scales. Version 0.3.9 of Whiley's compiler was used to compile the code for comparison. This is not the latest version, but it is the version which the solution for this project is built on. There are not significant changes to the concurrency model between the two versions for this difference to have an impact on the results.

There is significant overhead in starting the JVM, and it employs a number of techniques to speed up routines that occur often [12]. To overcome the discrepancies these factors might introduce into the results the tests are both run and tested from within the JVM, and are run

10 times before any results are recorded. To avoid noise in the results, the test is run 50 times for each input (this does not include the pre-recording runs) and the measurements are averaged. All measurements are made in milliseconds.

The first benchmark is the 'matrix' program. This takes two square matrices of equal size and multiplies them in a concurrent way. The algorithm spawns the square of a matrix's size in actors to solve the problem: if their size is 10x10 then 100 actors will be used to solve the problem. This is a particularly useful benchmark because the number of actors scales up significantly as the input increases. Starting at an input size of 5x5, the size was increased by another 5 until either the program crashed or the matrices reached 100x100.

The second benchmark is the 'sum' program. This takes a list of numbers and computes their sum in a concurrent way. For every 100 numbers it spawns an actor and then makes them perform the sum of their 100 numbers in parallel. Once this is complete, it places the results into a new list and runs the algorithm again until the list contains only one item. For this benchmark the number of actors is roughly proportional to the input size, but larger input sizes will skew the number of actors upward. Starting at an input size of 100, the size was increased by another 100 until 25,000 was reached.

The source code for each benchmark can be found at `https://github.com/DavePearce/wybench/tree/master/concurrent/micro`.

5.2.2 Results

Although the averaging of several results hasn't always accounted for noise in the measurements, the results of the benchmark unanimously confirm the prediction that thread overheads would slow down the existing solution. As the number of actors increases the difference in speeds between the two solutions becomes more pronounced as the overheads become more and more prevalent.

It is the matrix benchmark that shows the success of the project. As the input size increases the difference in speeds becomes more pronounced, until the existing implementation is overcome by memory limitations and ceases to even function. The server in particular, with its minimal memory size, barely has a visible plot for the threads. The new solution is more than capable of continuing for much greater input sizes — see figures 5.1, 5.2 and 5.3.

This makes the results of the sum benchmark more interesting. This benchmark deals exclusively with small numbers of actors, and the existing implementation is consistently faster by a small margin all the way up the scale, in figures 5.4, 5.5 and 5.6. It's likely that the increased parallelism of many threads at this size are overcoming the overheads for a equation that responds well to concurrency.

5.2.3 Analysis

The results of the matrix benchmark are not particularly surprising, considering the prediction at the start of this chapter. In fact, they confirm the hypothesis that this project is founded on. Mapping actors intended for mass use onto heavy threads is not a reasonable solution.

The results of the sum benchmark are of the more concerning nature, as the new solution is consistently slower than the existing one. The difference in performance is likely attributable to the unoptimised use of yielding, though. Yielding is not a small operation, even for toy examples. More analysis is required to see where performing a yield produces better performance than not doing so.

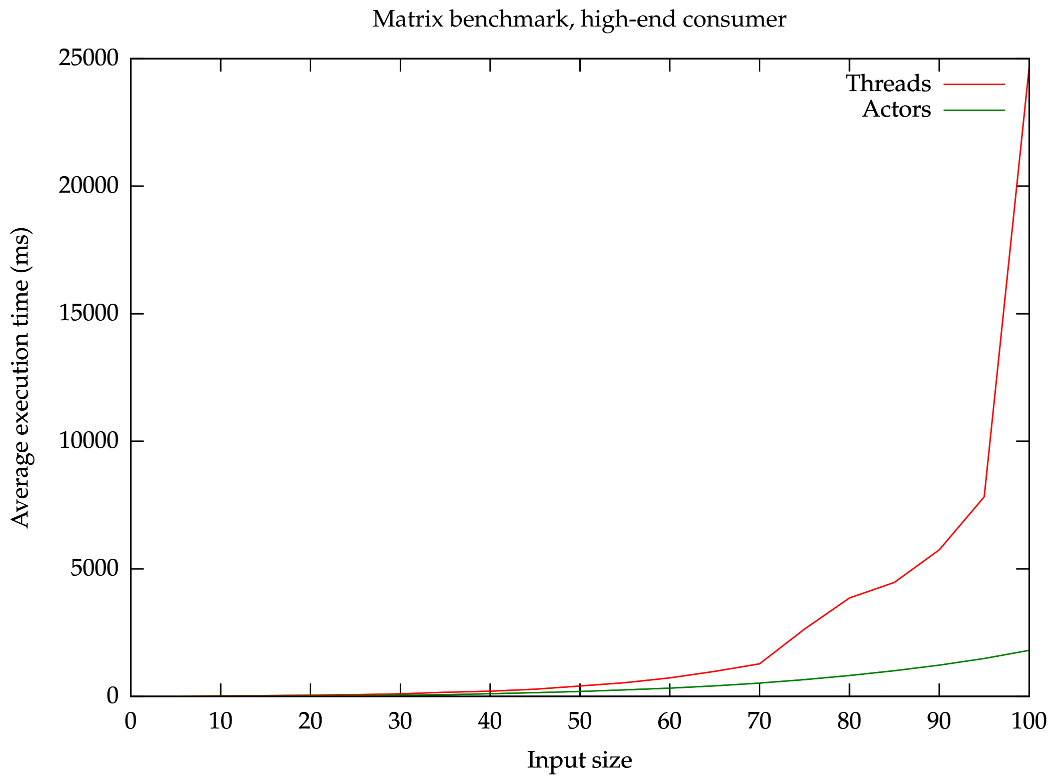

Figure 5.1: The results for the matrix benchmark on the desktop

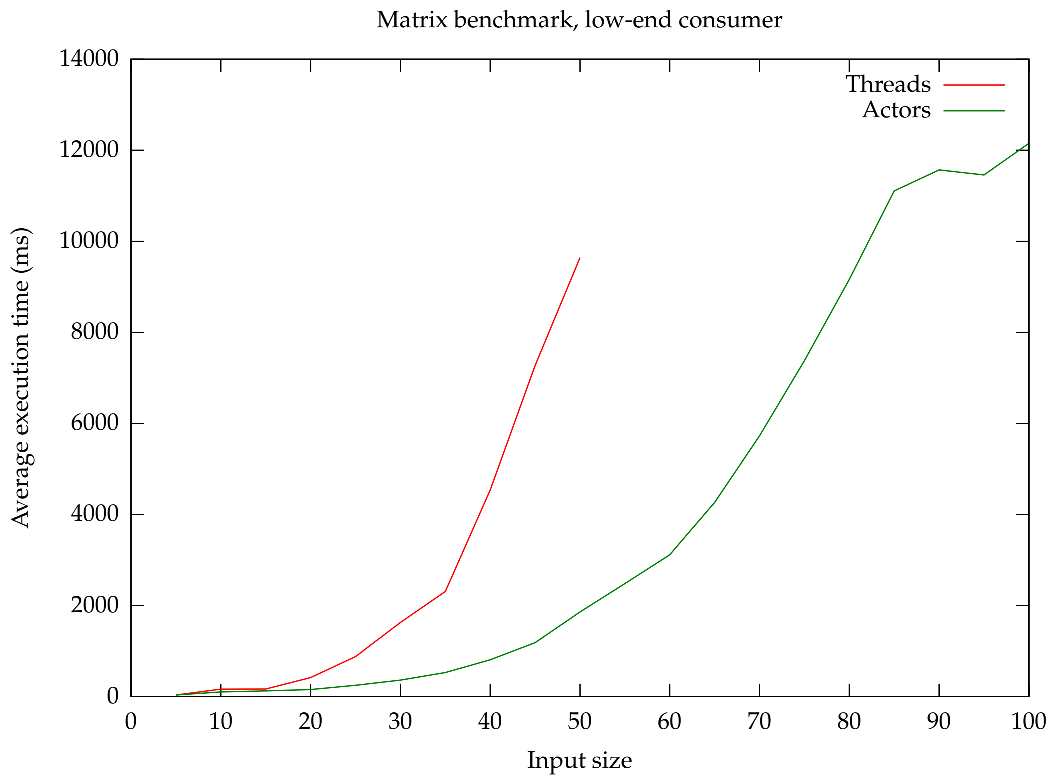

Figure 5.2: The results for the matrix benchmark on the laptop

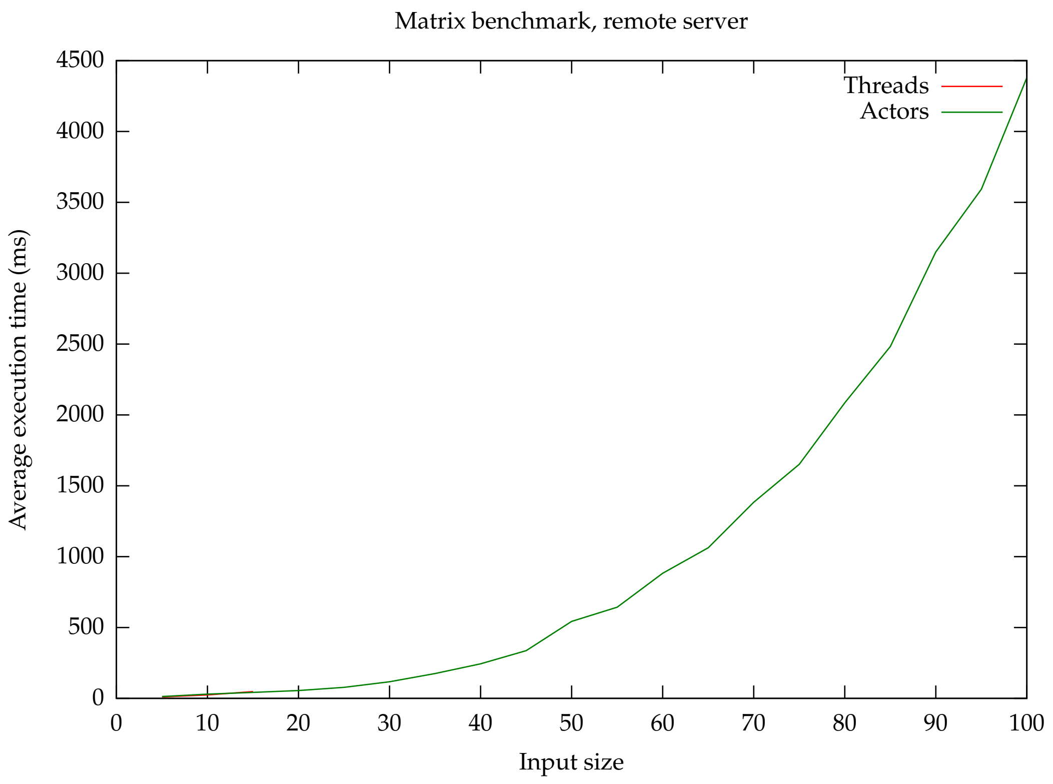

Figure 5.3: The results for the matrix benchmark on the server

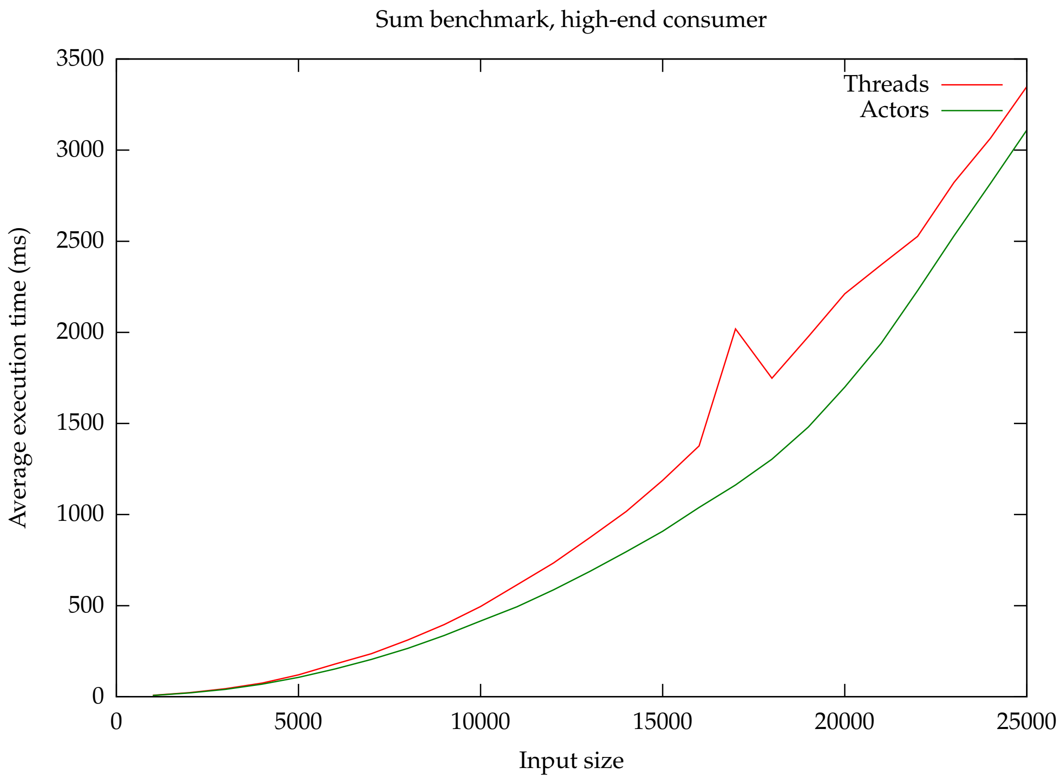

Figure 5.4: The results for the sum benchmark on the desktop

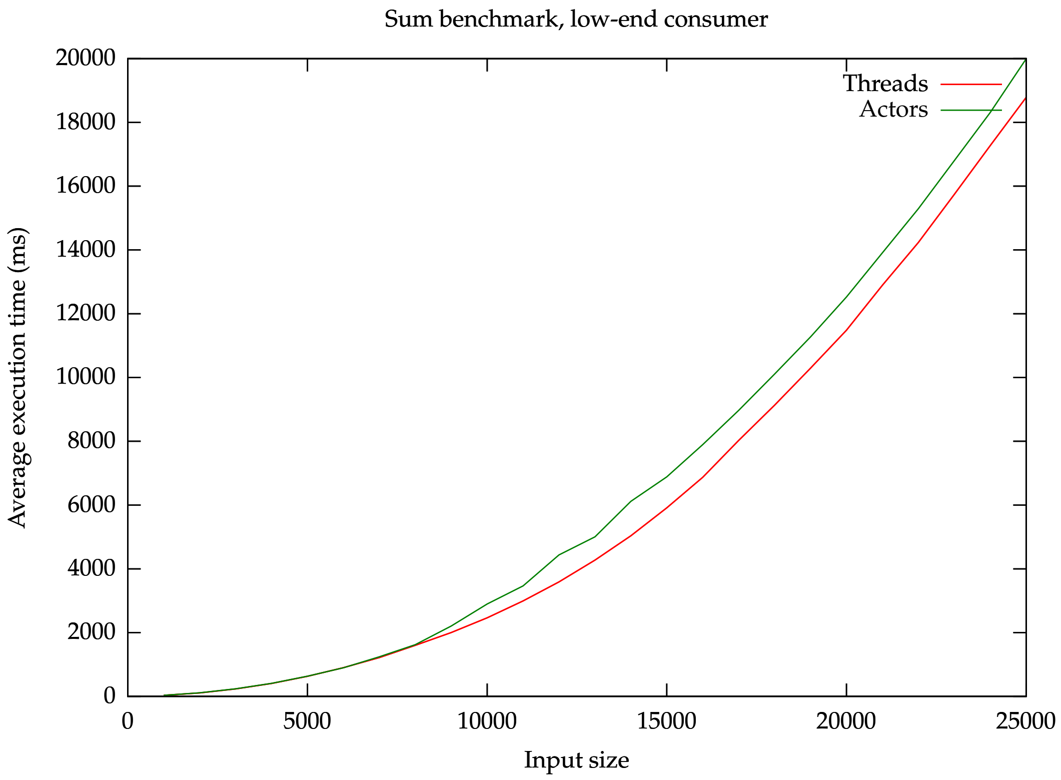

Figure 5.5: The results for the sum benchmark on the laptop

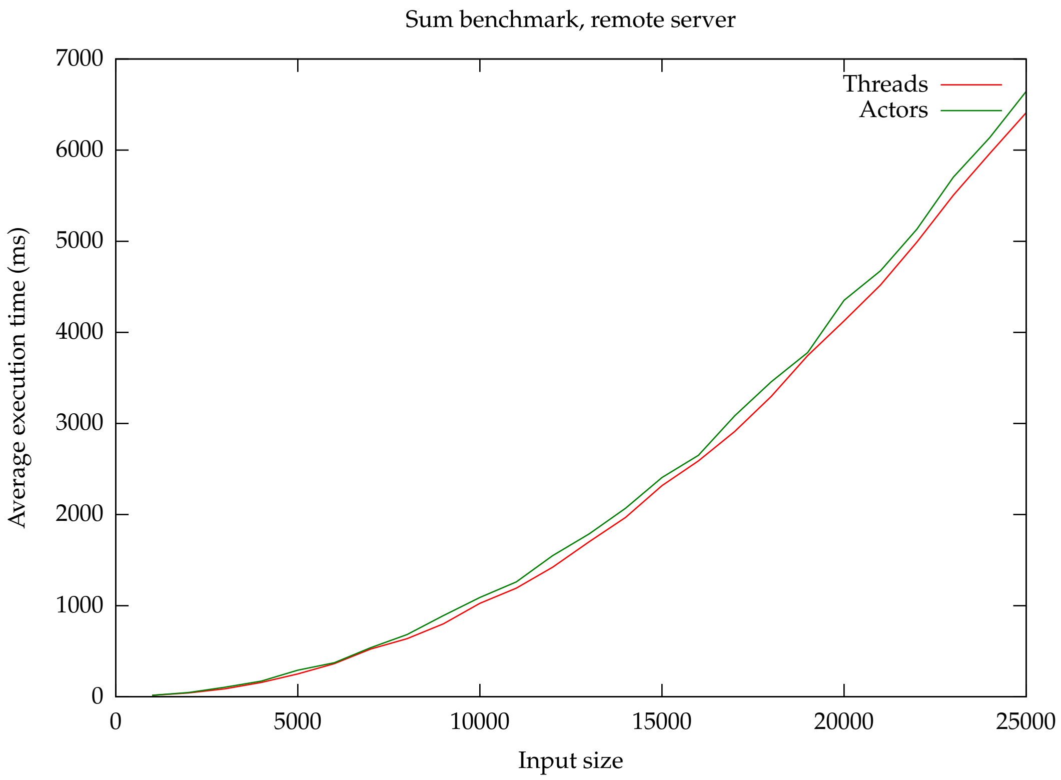

Figure 5.6: The results for the sum benchmark on the server

Of course, it is expected that mapping each actor to a thread will generally be more performant at these sizes. The overheads are much less of a problem, and the new solution's fixed-size thread pool will produce overhead of a similar amount. The only way to achieve comparable speeds for this range of actors is to optimise away the difference. Different techniques have been identified and are discussed in the next chapter.

Nevertheless the project has achieved its goal of supplanting the existing implementation because of the unacceptable performance drop and ultimate memory problems it has at only a middling number of actors. The system is moderately effecient and ripe for optimisation alongside the core language.

Chapter 6

Future Work

“Premature optimization is the root of all evil in programming.”

— C.A.R. Hoare

During development a number of points that could be optimised were identified. They were not attempted at the time to avoid introducing unnecessary problems, and some of these points are marked in the code for future work. Here is a list of optimisations that could be made in the future as well as general work that still needs to be done:

- While the ability to choose whether or not to yield the sender of a message is given to the receiver at runtime, at the moment the choice is static, decided at compile-time. This behaviour should be updated with a more sensible, context based choice.
- The type flow algorithm for the bytecode rewriting doesn't take into account that if local variables are not used after the point the code yields at, there is no need to save them. This means continuations take longer and use up more memory than is needed.
- An 'entry method' like `System::main` is considered an asynchronous message send and so if it throws an exception there is no reaction. I've refrained from implementing this because a later version of Whiley will standardise entry points around headless methods, and this is where that belongs. For now, if any message fails it prints the stack trace of the cause.
- The compiler has the ability to specify the number of threads in the thread pool, but only at compile-time. It might make sense to allow a Whiley program to modify this at runtime to adapt to particular situations or hardware. It's not possible to modify the number of threads in a pool of fixed size, though.
- Headless methods don't operate on actors, but they still need to be concurrent. Adding 'strands' (actors without state) involves modifying a part of the compiler outside of this project's components, and would involve a greater understanding of the compiler's inner workings than was necessary.
- Whether or not the pure function yield needs to be decided. If they take some time to evaluate then they will monopolise the thread, but they can't send messages so the current implementation would never make them yield.

Chapter 7

Conclusion

This project has implemented the actor model on the Java Virtual Machine for the Whiley programming language. The JVM provided many challenges for the project, as it supports only heavy threads as a form of concurrency. The implementation made use of practical techniques like continuations and yielding, and created both runtime and compile-time components for the solution. The compile-time components perform type flow analysis and Java bytecode rewriting to add functionality not inherent to the JVM, while the runtime components manage the concurrent interactions of the actors.

The evaluation provided useful results about the success of the solution. There is still work to be done on optimising the rewriting algorithms, and several points in the code where this could be achieved have been identified. For large numbers of actors — a situation which is an expectation of the actor model — this solution achieves the desired functionality and outperforms Java's heavy threads. While the software's performance is not on par with more mature solutions, its integration into the Whiley compiler will allow greater optimisation for the specifics of the language.

Due to the success of the project, the solution will soon be merged with the core Whiley compiler. This means that others will be using it to achieve concurrency in their own work. Whiley is still an ongoing project and so it is expected that this solution will evolve alongside it. As the performance of Whiley improves so too must this solution, and I personally intend to continue development on it if possible.

Bibliography

[1] AGHA, G. A. Actors: A model of concurrent computation in distributed systems. Tech. Rep. 844, MIT Artificial Intelligence Laboratory, June 1986.

[2] ARMSTRONG, J. Erlang. In *Communications of the ACM* (Sept. 2010).

[3] ARMSTRONG, J., WILLIAMS, D., AND VIRDING, R. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[4] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] GROSSMAN, D. The transactional memory / garbage collection analogy. In *Proc. OOPSLA'07* (Oct. 2007).

[6] HICKEY, R. Clojure. `http://clojure.org/`.

[7] JAMALI, N., AND REN, S. A layered architecture for real-time distributed multi-agent systems. In *Proc. ICSE'05* (May 2005).

[8] KARMANI, R. K., SHALI, A., AND AGHA, G. Actor frameworks for the jvm platform: A comparative analysis. In *Proc. PPPJ'09* (Aug. 2009).

[9] KNUTH, D. E. *The Art of Computer Programming*, 3 ed., vol. 1, Fundamental Algorithms. Addison-Wesley, 1997, ch. 1.4.2: Coroutines, pp. 193–200.

[10] LAUTERBURG, S., DOTTA, M., MARINOV, D., AND AGHA, G. A framework for state-space exploration of java-based actor programs. In *Proc. IEEE/ACM International Conference on Automated Software Engineering* (2009).

[11] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*, second ed. Sun Microsystems, Inc., 1999.

[12] ORACLE. Java se hotspot at a glance. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`.

[13] PEARCE, D. J. Whiley: an open source programming language with extended static checking. `http://whiley.org`.

[14] PEARCE, D. J., AND NOBLE, J. Implementing a language with flow-sensitive and structural typing on the jvm. In *Proc. The Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE), 2011* (2011).

[15] PYTHON SOFTWARE FOUNDATION. The Jython Project. `http://www.jython.org`.

[16] SALCIANU, A., AND RINARD, M. Purity and side effect analysis for java programs. In *Proc. VMCAI* (2005), pp. 199–215.

[17] STRACHEY, C., AND WADSWORTH, C. P. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation 13* (2000), 135–152.

[18] THE JRUBY TEAM. JRuby. `http://jruby.org/`.

[19] THIELECKE, H. Continuations, functions and jumps. In *Logic Column, SIGACT News* (1999), J. G. Riecke, Ed.

[20] TRIFORK A/S. Erjang. `https://github.com/trifork/erjang/wiki`.