A LINGUISTIC APPROACH TO CONCURRENT, DISTRIBUTED, AND ADAPTIVE PROGRAMMING ACROSS HETEROGENEOUS PLATFORMS

PAUL HARVEY

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

OCTOBER 5, 2015

Abstract

Two major trends in computing hardware during the last decade have been an increase in the number of processing cores found in individual computer hardware platforms and an ubiquity of distributed, heterogeneous systems. Together, these changes can improve not only the performance of a range of applications, but the types of applications that can be created.

Despite the advances in hardware technology, advances in programming of such systems has not kept pace. Traditional concurrent programming has always been challenging, and is only set to be come more so as the level of hardware concurrency increases. The different hardware platforms which make up heterogeneous systems come with domain-specific programming models, which are not designed to interact, or take into account the different resource-constraints present across different hardware devices, motivating a need for runtime reconfiguration or adaptation.

This dissertation investigates the actor model of computation as an appropriate abstraction to address the issues present in programming concurrent, distributed, and adaptive applications across different scales and types of computing hardware. Given the limitations of other approaches, this dissertation describes a new actor-based programming language (Ensemble) and its runtime to address these challenges. The goal of this language is to enable non-specialist programmers to take advantage of parallel, distributed, and adaptive programming without the programmer requiring in-depth knowledge of hardware architectures or software frameworks. There is also a description of the design and implementation of the runtime system which executes Ensemble applications across a range of heterogeneous platforms.

To show the suitability of the actor-based abstraction in creating applications for such systems, the language and runtime were evaluated in terms of linguistic complexity and performance. These evaluations covered programming embedded, concurrent, distributed, and adaptable applications, as well as combinations thereof. The results show that the actor provides an objectively simple way to program such systems without sacrificing performance.

Acknowledgements

Behind the words in this document there are many years of stories. The following is an acknowledgement and thank you to the people who helped make this the case.

My thanks and appreciation go to my supervisor Joseph Sventek who guided the work in this dissertation. I have learned a lot from him and had fun in the process. Also to my second supervisor Wim Vanderbauwhede, who offered many helpful and different perspectives on this work, as well as many entertaining conversations about Japan.

During my PhD, I had the opportunity to work at the University of Aizu in Japan. My thanks go to Saji Hameed for being colleague, caretaker, and tour guide during my stay.

I am grateful to the Carnegie Trust for the Universities of Scotland for funding this research.

To my friends who took the time to let me annoy them: Balir Archibald, Callum Cameron, Martin Ellis, Kristian Hentschel, Paul Jackma, Simon Jouet, Alexandros Koliousis, Fergus Lehey, Craig McGluaghlin, Magnus Morton, Syed Waqar Nabi, Lauren Norrie, Stephen Strowes, and Kyle White.

To my family, for the love, support, and encouragement that they gave. Especially from my little brother and his motivational pictures of champagne and sunsets.

And finally, but most deservedly, to my best friend, co-adventurer, and partner Phillipa. I am eternally grateful for your love, support, kindness, and occasional but necessary kick up the arse which saw me through this, as well as having the odd adventure along the way.

“Don’t you see that the whole aim of Newspeak is to narrow the range of thought? In the end we shall make thought-crime literally impossible, because there will be no words in which to express it. Every concept that can ever be needed will be expressed by exactly one word, with its meaning rigidly defined and all its subsidiary meanings rubbed out and forgotten.”

— George Orwell, 1984
On the power of controlling expression.

Contents

List of Tables

List of Figures

Acronyms

ABC Assignments, Branches, and Conditions.

AFAPL Agent Factory Agent Programming Language.

AST Abstract Syntax Tree.

BRS Bigraphical Reactive Systems.

CELF Compact Executable and Linking Format.

CFG Control Flow Graphs.

CG Call Graph.

CPU Central Processing Units.

ELF Executable and Linkable Format.

EVM Ensemble Virtual Machine.

FPGA Field-Programmable Gate Array.

GPU Graphical Processing Units.

HPC High Performance Computing.

IR Intermediate Representation.

JIT Just In Time.

JVM Java Virtual Machine.

PIC Position Independent Code.

RSSI Received Signal Strength Indication.

UUID Universally Unique ID.

VM Virtual Machine.

WSN Wireless Sensor Networks.

Chapter 1

Introduction

Single core Central Processing Units (CPU) were once the champions of increasing application performance. This was achieved by increasing the number of instructions executed per cycle, increasing the depth of pipelines, and performing more speculative execution, granting more instruction level parallelism at higher clock speeds [1, 2]. From the early 2000s, clock propagation delay, heat dissipation, energy constraints, and the *memory wall* [3] have meant that single core CPUs are no longer the champions of increasing application performance as their clock speeds are no longer increasing, and the practical limits of instruction level parallelism have been reached for sequential applications. Instead, hardware developers have turned to an army of champions. This has been realised in two ways: an increase in the number of CPU cores per hardware device, such as multicore CPUs and Graphical Processing Units (GPU), and the ubiquity of distributed systems composed of heterogeneous hardware devices - i.e. the many different types of connected computer in the world today.

Multicore processors are capable of concurrently executing many threads of control simultaneously, requiring developers to create applications accordingly. Traditional concurrent programming relies on multiple threads of execution which share state. This approach often leads to programming errors, such as data race conditions and deadlock. As the number of cores is set to increase to the hundreds and beyond, the threaded approach to programming simply does not scale.

As well as increasing the number of processing cores locally, improved battery technology and low-power micro-controllers have enabled distributed systems of heterogeneous devices to become embedded in the world around us. These systems consist of many different types of device, from small battery-powered sensors, to mobile phones and tablets. This range of devices represents different levels of computing scale, with many different operating conditions, resource constraints, and programming models.

The goal of this work is to ease the burden of programming these concurrent and heterogeneous distributed systems such that they can be used by non-specialist programmers. This work shows that using an actor-based abstraction to create applications for either individual systems, or collections of such systems, not only provides a natural way to express solutions to the problems of such programmers, but does so in a way which is simpler than existing techniques and easily facilitates new programming options in the general case, such as runtime application reconfiguration, or *adaptation*. Here, adaptation refers to the ability for software to change the topology of connections between communicating entities, and either install or migrate executing software to remote locations at runtime. Additionally, moving some of this burden into a runtime which supports the actor-based abstraction, further simplifies the task of the programmer. By presenting a more appropriate programming model, and pushing complexity into the runtime, this work aims to increase the use of concurrent and heterogeneous distributed systems.

This goal is to show that by creating applications which are composed of encapsulated actors which communicate by explicit message passing, not only can one address existing problems in programming such systems, but one enables a number of these systems to be programmed collectively. To prove this assertion, this dissertation describe the design and implementation of an actor-based programming language and runtime system. Using these tools, the actor abstraction is applied to programming embedded devices, highly concurrent devices, and adaptive programming of heterogeneous devices at different levels of computing scale.

1.1 Thesis Statement

Hypothesis : *The use of encapsulated, shared-nothing loci of computation and explicit message passing, found in the actor programming model, will both enable and simplify the programming of concurrent, distributed, and adaptive applications across heterogeneous platforms at different levels of computing scale.*

This assertion will be demonstrated by

- The creation of a general purpose actor language, with the actor as the unit of adaptation and concurrency
- The creation of a compiler for the actor language that translates applications into an intermediate language, which is executed by a virtual machine on different classes of hardware platforms, and enabling actor adaptation across the scale space.
- An evaluation of the application of the actor-based abstraction to embedded programming.
- An evaluation of the application of the actor-based abstraction to kernel-based programming.

- An evaluation of the application of the actor-based abstraction to adaptive programming.

1.2 Contributions

This work contributes to the abstraction of programming concurrent, distributed, and adaptive applications in the following ways:

- The development of a new actor-based programming language which natively supports the discovery and reconfiguration of actors at runtime, as well as a channel-based abstraction of the network medium.
- The development of a lightweight runtime to execute actor-based applications on a number of platforms, including highly-constrained, embedded devices.
- A simple type mechanism and compiletime analysis to minimise memory consumption in the context of a shared nothing environment.
- An investigation of the impact of actor-based programming on embedded programming in terms of linguistic complexity and application performance.
- The first in-language application of actors to accelerator-based programming of parallel devices as well as an investigation of the impact of actor-based programming on accelerator-based programming in terms of linguistic complexity and application performance.
- An exploration of the impact of actor-based programming on adaptive programming in terms of linguistic complexity and application performance

1.3 Publications

The work reported in this dissertation has led to the following publications:

- *"Parallel Programming in Actor-Based Applications via OpenCL"*
  P. Harvey, K. Hentschel, and J. Sventek
  16th International Conference on Middleware (to appear)
- *"A Virtual Machine for the Insense Language"*
  C. Cameron, P. Harvey, and J. Sventek
  6th International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications

- *"Channel and Active Component Abstractions for WSN Programming: A Language Model with Operating System Support"*
  P. Harvey, A. Dearle, J. Lewis, and J. Sventek
  1st International Conference on Sensor Networks

During the work on this dissertation, the following papers were also published by this author on related topics:

- *"Accelerating Lagrangian Particle Dispersion in the Atmosphere with OpenCL"*
  P. Harvey, S. Hameed, and W. Vanderbauhede
  2nd International Workshop on OpenCL

- *"Wireless Sensor Network Simulation With Xen"*
  P. Harvey, and J. Sventek
  The 46th Annual Simulation Symposium

1.4 Projects

Given the amount of implementation required to explore the hypothesis, a number of projects were carried out by undergraduate students which contributed to this work. The background, motivation, ideas, proposal, and supervision of these projects were carried out by the author, however, the implementation was completed by students. The following is a list of the projects and students:

- Callum Cameron : The implementation of the Ensemble VM on the Tmote Sky embedded platform.

- Kristian Henstchel : The extension of the Ensemble language to use an OpenCL library in Java.

- Craig McLaughlin : The addition of the movable type and compiletime analysis to the Ensemble type system.

In each case, the implementation from the student project was integrated into the main project, and then expanded upon by the author.

1.5 Outline

The dissertation is structured as follows:

Chapter 2: covers related works in the areas of actor languages, adaptation methodologies and implementations, parallel programming approaches, embedded programming styles, and compiletime analysis.

Chapter 3: describes the design of the Ensemble language which is used as the mechanism to explore the hypothesis.

Chapter 4: chronicles the design and implementation of the Ensemble Virtual Machine, including support for runtime discovery and adaptation of actors and stages across multiple hardware platforms at different levels of computing scale.

Chapter 5: argues that the actor is the appropriate programming mechanism for concurrency, distribution, and adaptation. This is evaluated in three different areas: embedded programming, accelerator-based concurrency, and multi-platform adaptation.

Chapter 6: summarises the points made in the dissertation and discusses future directions for the work.

Additionally, there are three appendices:

Appendix A provides a formalised description of the compiletime analysis associated with the movable type described in Section 3.2.

Appendix B describes the format of the modified Java class files which are generated from the Ensemble linker, and executed by the Ensemble VM.

Appendix C presents the results on performance and space consumption of native Ensemble applications on embedded hardware. This appendix also presents the microbenchmark results for the performance of the individual adaptation operations in Ensemble.

Chapter 2

Related Work

The increase in the number of processing cores in modern CPUs, coupled with the ubiquity of distributed systems composed of heterogeneous devices, has led to a set of disparate and unconventional programming styles required to program such devices and systems. As well as dealing with the inherent issues of multithreaded programming, such as deadlock and scalability, programmers must now also consider the resource constraints and networking capabilities of different hardware platforms. As these technologies are used more and more by non-specialist programmers to solve a wide range of problems, the challenge of programming such systems is even more problematic.

The goal of this dissertation is to reduce the burden of programming such systems via the abstractions presented by the actor model of computation. Although the issues associated with programming these systems must be considered collectively, for discussion they can be partitioned into four equivalence classes: concurrent programming, embedded programming, distributed programming, and adaptive programming. To discuss the related work in these areas, this chapter is arranged as follows. Section 2.1 provides an overview of a number of different actor languages. Section 2.2 discusses motivations and technologies with regards to runtime adaptation of software, at different levels of computing scale. A summary of the challenges of programming parallel devices is presented in Section 2.3. An overview of relevant program analysis approaches in given in Section 2.4, and Section 2.5 summarises the points made.

2.1 Actor Languages

One programming model which addresses programming concurrent and distributed systems is the actor-model of computation. Applications using this model are expressed in terms of *actors*. An actor is a self-contained entity with private state and its own locus of control, usually a thread. An actor communicates with other actors via message passing, often (but not exclusively) in an asynchronous fashion. The actor model was first proposed by Hewitt et al. [4] as a model of independent execution and communication. Agha [5] extended the model, introducing mailboxes to store sent messages for later consumption. Without mutable shared state, actors enable a safe programming model for concurrency, with good performance [6]. Deadlock and race conditions caused by locking mechanisms around shared state cannot occur as there is no shared state or locking mechanism. Furthermore, the use of message passing between actors provides an abstraction that is transparent to the location of the actors involved. Therefore, message passing can transparently be used for communication between actors on the same hardware, or over a network. Such abstraction of location is known as location transparency.

Despite the substantial advantages offered by the actor model, there are a number of trade-offs when compared to languages, such as C, which should be considered. Firstly, the lack of shared state introduces overhead in the language, and potentially the runtime. Rather than simply enabling two threads access to a shared array, the actor model requires this array to be sent between them. For developers writing performance critical code, who are comfortable with the potential issues, the use of shared memory may be more appropriate. Secondly, the infrastructure code which is required to facilitate inter-actor communication may be too verbose for simple applications, where scripting approaches may be more suitable. Thirdly, the peering relationship between actors does not by default support an indication of the priority of a message sent between actors - there is no default way to distinguish between messages of a high or low priority. However, individual implementations of the actor model have methods for coping with this, such as multiple channels being used to create priority hierarchies.

Since the actor-model was first described, there have been a number of languages which are based on this model. It would not be possible to provide a discussion of all actor languages, instead, representative examples from each equivalence class is made in this section. Of the actor languages and systems discussed in the literature, three equivalence classes are present:

- Languages which are targeted at particular problem sets.
- Languages which are embedded in other languages or systems.
- Languages which are for research purposes.

The following shows that the existing approaches are not sufficient to prove the hypothesis.

Niche Problem Sets

Erlang [7, 8] is a well-known programming language and runtime system developed in 1986. It mixes functional programming and an actor based model of concurrency. Actors exchange asynchronous messages to communicate information with each actor possessing a message buffer (known as a mailbox) to hold messages until they are processed. It also supports *hot swapping* (or replacement) at the granularity of a function. Adaptation in Erlang consists of exchanging the functionality of how a received message is processed, rather than the loadable modules or code migration discussed in Section 2.2.1. Erlang supports location transparent communication between actors. Remote actors are either explicitly bound to another actor by knowing the remote actor's address, or the remote actor is spawned and then bound to the spawning actor. In order to deal with the unreliable nature of communication between remote actors, Erlang uses a combination of exceptions and runtime support following the *let it crash* philosophy. Remote actors are linked together either implicitly when they are spawned, or explicitly if being connected to, and when a remote actor generates an uncaught exception it kills itself and propagates a kill message to the actors to which it is connected. They, in turn, kill themselves and propagate the message to the actors to which they are connected. This continues until all actors are dead or the kill message is caught and handled. The assumption is that the actor which catches the message is able to enact recovery.

While Erlang is a very mature language and runtime and is well suited to the domain for which it is designed (telecommunication systems), there are some shortcomings. There is no syntactic notion of a process, thread, or actor. Instead there are modules with collections of functions. As the size of an application grows, the lack of clarity within the language makes comprehension difficult when trying to determine the behaviour of the application. Also, the fine grained *hot swapping* makes adaptation challenging, requiring the user to have detailed knowledge about what code should be relocated at the function level, rather than the process/thread/actor level. Erlang does not natively support process migration, and there is no provision in the language for it. Also, in terms of scale the smallest device that can successfully run an Erlang application is the RaspberryPi[1] (see Section 4.3.3). This platform has approximately 4 orders of magnitude more RAM than the smallest platform targeted by this work (Section 4.3.2).

ActorNet is an actor language and system for Wireless Sensor Networks (WSN) built on top of TinyOS. It has the ability to relocate an actor from one small battery-powered computer used in WSNs (mote) to another as actors are interpreted. Actors are programmed in a language based on the functional language Scheme [9], with appropriate extensions for actors, such as the `send` keyword to facilitate message passing. Little or no detail is given on the semantics of message passing including discovery and binding between actors. The work claims that this language is simple, however each actor is specified as a number of functions and lambda expressions, resulting in a non-linear flow of code. It is also the case that the language does not present an intuitive abstraction over hardware, using static numbers to reference hardware devices: `io 0`. This language is interpreted by an interpreter running

[1]http://www.erlang-embedded.com/ - Accessed October 2013

on a TinyOS mote. The runtime system is multi-threaded with stack-based threads, however as TinyOS is event-based, allocation and scheduling of these threads is done within the runtime. This is in contrast to the static preallocation approach used in TinyOS by default, Section 2.2.1. Coexistence of two such different abstractions is not efficient in space or power, especially when not implemented within the kernel [10], and there are no detailed results on the effect of these features. ActorNet provides a virtual memory system using external flash to enable more runtime memory which is required to be able to interpret the applications, however the only listed applications perform simple operations such as printing. It is unclear how much space a more complex application would require. ActorNet is only implemented for the Mica2 hardware platform[2]. No work has been done on ActorNet since 2005. There is no discussion about how the language or runtime deals with transparency in relation to *environmental entanglements*.

Environmental entanglement refers to the links between an actor (or software unit) and its current execution environment. Examples include bindings to other actors, both in terms of the actual bindings themselves, as well as the higher level interaction between actors. Consider an actor A which has a file open on a given machine. Should this actor be migrated to another machine, what should happen to the file and the actor's link to that file? Should the file be copied, or moved entirely, should the actor have a remote reference to the file, or should the actor not be allowed to migrate in this situation? Environmental entanglement is a challenging issue and influences the design and implementation of languages and their runtimes.

JoCaml [11] is a system for mobile agents built inside of the Object-Caml (Ocaml) language. Although not strictly an actor language, the work broadly has similar goals to this thesis. The language has the ability to create uni-directional, typed channels, as well as being able to remotely instantiate or migrate processes; in this context, a process refers to a thread. The language aims to be *simple, expressive* and *consistent* such that JoCaml applications are location transparent in a way that is understandable.

JoCaml channels are different from Ensemble channels, as when defining a channel both end points of the channel are declared, as in Rust (Section 2.4.1), making dynamic runtime binding of channels impossible. Also, channels do not simply convey messages, instead, they accept data and then invoke code to process this data at the receiver process. This requires that upon receipt of a new message a new thread is created at the receiver. This would not be a feasible approach on a resource-constrained device.

Axum [3] (codename Maestro) is a domain-specific actor-based programming language de-

[2]http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf - Accessed July 2014

[3]channel9.msdn.com/shows/Going+Deep/Maestro-A-Managed-Domain-Specific-Language-For-Concurrent-Programming/ - Accessed November 2012

veloped by Microsoft. It is domain-specific in that it is intended for use in highly concurrent applications. Actors communicate via locally synchronous and remotely asynchronous message passing. Although not directly discussed, it is assumed that this design choice in Axum, and other systems [12, 13], is made because function/method calls are used for local message passing, and network-based packet transmission is used for remote message passing. Actors may be grouped into domains to enable direct sharing of actor state between actors within the domain. To facilitate message passing, an actor (Agent in Axum parlance) has a channel associated with it. Channels are complex data types, containing multiple unidirectional buffers, functions, and even communication protocols specified via state machines. There is no discussion of how network failures are managed, if migration is a design consideration, or the implication of how the sharing of state between domains, requiring locking primitives, would not undermine the point of using *shared-nothing* semantics. There is little documentation of Axum other than blog posts. Work on the project was discontinued in 2010[4].

Embedded Languages

Scala [14, 15] is a popular functional language. Scala applications are interpreted by the Java Virtual Machine (JVM). One of the main goals of Scala is to fuse functional and object orientated programming, where functions are available as a part of objects, rather than objects in their own right. The fusion of styles has met with mixed success [16].

Within Scala, concurrency is not an inherent part of the language, and is provided by libraries that offer actor-based concurrency. There are a number of different implementations, currently the akka[5] actor framework is the most popular. It addresses a number of limitations of the previous actor implementation and offers a verbose, configurable setup.

Actors have private state and interact via asynchronous message passing. Synchronous message passing is also possible via asynchronous message passing and futures in user level applications, where logic will block on the result of the future to ensure that the message has been delivered. In terms of distributed computing, Scala, like Erlang, is able to communicate with or spawn *remote actors*, although unlike Erlang, the error mechanism exclusively uses Java-like exceptions. Within Scala there is no first class location type, instead, remote actors are instantiated with IP addresses, and then references to these actors are used. Like Erlang, there is the ability to hot swap code, however migration is not provided by the language. The language does support gaining references to other actors by specifying a Unix style path to the actor, however, this requires knowledge of the actor's location, and is not a generic

[4]msdn.microsoft.com/Forums/en-US/axum/thread/ae809d82-42ba-42bb-9199-e1e9489a82fe - Accessed November 2012

[5]http://akka.io/ - Accessed October 2013

approach. Due to the presence of all of these runtime features, the Scala code base is quite large. The Scala for Android project[6] highlights that ignoring the OS and Java runtime, the scala runtime occupies 8MB of storage. Conditional compilation can be used to reduce this to 25KB of storage, with the hello world application requiring 10KB of storage. This represents almost 15% of the available storage on the smallest platform targeted by this work, and represents a static application. In terms of RAM, if Scala were to conform to the J2ME specification, it would still require 3 orders of magnitude more RAM than is available on the smallest platform targeted by this work.

Salsa [17] is an actor-based programming language embedded within Java. Salsa applications are preprocessed to generate standard (or *vanilla*) Java applications which are compiled in the normal way. By embedding itself within Java without any special runtime requirements, Salsa is automatically available on many JVM supported platforms.

Salsa is embedded within Java, using a pre-parser to generate vanilla Java code. While this provides Salsa with the large number of libraries Java has to offer, there is no list which specifies which of these are Salsa safe, in the context of the actor semantics of the language. This raises another point, that the Salsa compiler does not prevent or warn of inappropriate interaction with Java applications. This could lead to violation of the state encapsulation of actors. While Salsa does gain portability from the JVM, it is not clear if the restraints imposed on JVM implementations for embedded systems, discussed later, prevent Salsa being available on such platforms. As Salsa targets internet applications, this seems likely. The performance of Salsa applications is poor [18].

Another language within a language is **Stage** [18, 16]. Stage is an actor-based language which is embedded within Python. The goals of Stage are similar to this work, specifically, to address concurrency, distribution, and process mobility, although they do not attempt to address heterogeneous systems. Stage supports asynchronous, as well as synchronous message passing, however message passing is done via explicit `send` or `receive` methods on objects, where actors are represented as Python objects; unlike other languages in this section which are pre-processed in some way, Stage uses existing Python with extra functionality added to the runtime. Stage supports location transparency for communication, and *weak* actor migration (Section 2.2.5). Migration can occur between cores on a processor, or nodes on a network, however the former requires a Python interpreter to be running on each core, and the latter requires intelligence in the network to facilitate non-local migration, specifically, a trail of forwarding components to forwards message to actors which have migrated. Migration in Stage is expressed via callback functions, meaning that actors cannot "continue where they left off". The use of an event-based mechanism in this case leads to disjoint flows of control.

[6]https://code.google.com/p/scalaforandroid/ - Accessed October 2013

Considering the similarity of the goals between Stage and this work, it is an interesting piece of work, however there are some issues. Firstly, as it is embedded within Python, Stage programs use dynamic typing, and as such do not provide the programmer with a strict programming environment[7]. Stage has modified the Python runtime to enable features such as communication transparency, consequently, it is not clear how portable this work is. Also, apart from there being no further work on this project, there is no mention of how environmental entanglements are addressed.

The C++ actor framework (**CAF** [6]) is a C++ library providing actor-based concurrency. Using layers of C++ templates, developers perform template metaprogramming to create actors and explicitly send messages between them. The author's goals are similar to this work: using an actor-based abstraction to simplify programming of concurrent and distributed systems. Actors are represented by lightweight threads, and the framework supports its own scheduling system. The framework has shown good performance compared to other popular approaches of programming such systems in terms of performance and memory consumption.

By creating a framework in C++, CAF enjoys good performance, and easy integration with existing projects. However, despite the good performance results, there are a number of limitations of this approach compared to this work. Firstly, the use of template metaprogramming is required to take advantage of the popularity of C++, however its use is a non-trivial exercise [19], acting as a barrier to non-expert programmers. Also, while the framework does support an abstraction of inter-node communication via message passing, it does not support the discovery of other actors at runtime, and is not capable of supporting runtime adaptation because applications are compiled to static binaries. Also, to successfully marshal and demarshal user defined types for remote communication, developers must `register` these types. This is done automatically in this work by the language. Like this work, CAF supports an actor-based abstraction of OpenCL kernels (Section 3.3.1) for programming parallel hardware architectures, although unlike this work, CAF specifies kernels in terms of C strings. This means that they do not support kernels containing multi-dimensional arrays or nested references within structures, requiring developers to marshal and demarshal such data types themselves. Also, given the abstraction of the memory allocation and data movement used, it is not possible to leave data on a parallel device, a common optimisation used to greatly reduce execution time. Furthermore, as kernels are compiled by a separate compiler at runtime, error messages are less meaningful and delayed, unlike this work. CAF does not support applications on truly heterogeneous hardware platforms, only homogeneous hardware platforms. Finally, despite the desire of the authors to target the *Internet of Things*, they are currently working towards porting their system to a RaspberryPi, a platform with significantly more resources than the battery-powered embedded devices which are also targeted

[7]It should be noted that there are many opinions on this matter.

by this work.

Research Languages

Insense [20] is a component-based language designed to simplify WSN programming. Components are like actors, and communicate using explicit synchronous message passing along typed channels. It is possible to send a channel as a message, but it does not yet support sending a component as a message, as in the π-calculus [21]. Insense is not a functional language, instead presenting a reduced quasi-Java programming style. Insense does not support while loops, recursion, nested data types, dynamic arrays, or memory allocation within control structures or loops. Also, Insense does not support location transparent communication, local or remote discovery of types, and actors can not be relocated or migrated. Insense applications are executed by InceOS [22]. The Insense language and its runtime form the base upon which this work is built.

Emerald [23] is an object-based language, in which every programming entity from files to booleans are objects. The language enables objects to communicate by method invocation. This is the same for local or remote objects, thus the language supports location transparency. Although not an actor language with explicit channels, Emerald supports the ability to pass a `global` object reference from one object to another. If the receiving object is remote, the referenced object will be migrated. As the language directly supports process migration, a number of language constructs exist to assist with the task, specifically the ability to `fix`, `unfix`, and `move` code between defined `locations`. There was also the ability to `visit` a location. This operation would migrate to the location, execute, and then return. All of these features are specified in the language as keywords.

Emerald was designed when applications targeted specific machine hardware, and as a result the Emerald runtime was written entirely in C, relying on specific encodings of language types and stack formats to support code migration. It is unclear from the literature how dynamic communication topologies would be achieved without automatically causing code migration. Unlike other languages with well-defined communication channels, or mailboxes, when sending a global object reference between objects, this would cause the automatic migration of the component. Also, given that all entities in the language are objects, the space requirements are likely to be high, which is not good for embedded hardware, however there is no documentation to examine this.

The Actor-Based Concurrent Language (**ACBL**) refers to a family of languages:

- ABCL/1 [24] was the original incarnation of the language and is discussed here
- ABCL/R [25] and ABCL/R2 are the first and second generations of ABCL which use reflection and are a subset of ABCL/1

- ABCL/c+ [26] is a variant of ABCL/1 which is based on C rather than Lisp

The language was created to address the high degree of parallelism found in many different areas of computing, from A.I. to operating systems. The language is Lisp-like, and is based on objects with private state that interact via asynchronous message passing. ABCL/1 provides a select statement to make a guarded choice between messages. This work does something very similar, except the choice is between channels, not messages. Futures are provided to check on previously sent messages. It is also possible to use synchronous message passing.

This work was active between 1986 - 1990 and gives no examples of distributed programming or performance evaluation. Given lack of subsequent work, it is more focused on an academic exploration of the model, than deployment. However, it is worthy of note as one of the pioneers of the actor model.

H.A.L. is the High-level Actor Language [27], and is an experimental tool for parallel and distributed programming. It is an object-orientated language with an actor based concurrency model, which is compiled into C programs [28]. It supports both synchronous and asynchronous message passing, with synchronous message passing being built from asynchronous message passing. The language is based on Scheme. The work explores a number of interesting topics such as reflection and message forwarding. The language is built to execute on the CHARM [29] runtime, thus the language is available on multiple platforms without needing to have a multi-targeted compiler. Another interesting feature is that CHARM provides a distributed kernel, along with the issues involved with distributed garbage collection. However, such systems often lead to great complexity and challenges, as discussed in Section 2.2.5. The work hints at exceptions being present, but does not explicitly state this, nor does it give any real world examples, or performance results. HAL does not support runtime adaptation or discovery.

2.2 Adaptation Techniques

The ability and motivation to replace, relocate, and migrate software varies between different scales of computing device. Given the increased deployment and usage of such devices, as well as the different operating conditions and resource constraints that they present, it is now necessary to enable software executing on such devices to *adapt*.

Embedded systems, such as WSNs, find these features desirable due to the remote locations in which they are deployed. Data centres and High Performance Computing (HPC) clusters use these techniques to achieve load-balancing. These adaptation techniques were originally explored in the late 20^{th} century, however never widely proliferated. The following is an overview of the current state of the art in approaches to adaptation across a number of different computing scales. Based on this overview, Chapter 5 describes how an actor-based approach can be used to enable and simplify adaptation both within and between the different computing scales discussed.

2.2.1 Wireless Sensor Networks

There are a number of different techniques that are used in WSNs to replace or relocate software, namely binary updates, loadable modules, middleware, and virtual machines. These techniques are representative of embedded systems in general.

Binary Updates

The default in **TinyOS** [30] (and **Mantis** [31]) is image-based replacement. Here an entire binary file is sent via the radio and upon reception at the target node, the binary is saved to external flash. The existing binary is then rewritten, and the mote restarted. Should every mote in the network require update then no extra information is required, however version numbers are used to distinguish between motes who do and do not require update. Although applications for TinyOS are written in the component-based language nesC [32], the binary which is generated at compiletime has had all application structure removed. This is intentional as the compiler uses many optimisations such as whole program in-lining to reduce the size of the binary. The result is a highly optimised and highly coupled binary. Deluge [33] and XNP [34] are used to disseminate the whole binary image. As a replacement strategy, this method is safer in terms of the complexity of the update mechanism when compared to other techniques discussed below, however, it requires large amounts of power to transmit the entire binary via the radio.

A more efficient technique is binary differential patching [35, 36]. The essence of this approach is that the differences between the binary present on a mote and the new update binary will be sent to the mote. The existing binary will be patched and the mote restarted. This techniques requires less data to be sent than the full binary, however the data to be sent along with the required metadata leaves this method relatively energy expensive. Also, it requires the *diff* process to be aware of the current binary running on the mote to be updated. Both approaches require the mote to be restarted.

Loadable Modules

The goal of loadable modules is to replace or relocate a component (or module), providing a more fine grained way to adapt software. In this way, only a small amount of data needs be transmitted, and no reboot is required. There are a number of different incarnations of this approach.

Contiki [37] uses *dynamic linking* at runtime to support reprogramming [38]. At compile-time, the compiler generates Compact Executable and Linking Format (CELF) object files. These files are the same as normal Executable and Linkable Format (ELF) files, except that they contain 8 or 16-bit information, rather than 32 or 64-bit information. This is done as embedded systems have smaller word sizes than non-embedded systems, and reduces the amount of transmitted data. These files contain machine code, data, and *names* of functions. This code can not be executed until these names have been resolved into physical addresses. On the deployed node is a symbol table containing the names and locations of all functions. Upon reception of a CELF file, all symbol names are translated to addresses (*linked*) and the machine code is moved (*relocated*) into program memory. A detailed exploration of the power and space requirements for this and other techniques is documented [38], however adaptation is controlled via an API, and has no integration with the programming model. Also, this approach requires a complete symbol table to be present on the node at all times, and it is the developer's responsibility to handle an expected function not being present.

The **SOS** [39] kernel has a similar approach, however it uses Position Independent Code (PIC) which does not require the linking stage above. PIC uses relative references rather than absolute addresses, and does not require the symbolic linking as described above. To generate PIC, compiler support is needed. The benefit is a simpler reprogramming mechanism, however, not all platform architectures support PIC. Those platforms which do often place restrictions on the size of such code, and compiler support for such code is not wide spread [38].

Lorien [40, 41] is a component-based operating system focused on runtime relocation and replacement of components, but not migration. It is built upon the OpenCom component model [42]. As the system and application elements of Lorien are components themselves, most of the system is runtime configurable; the only exception is a small section of code which is used to manage reconfiguration. Interconnections between the components are achieved via interfaces, and each component specifies what it offers and requires. Thus, system integrity can be enforced at runtime by ensuring that components being removed, or inserted will meet the constraints. Essentially, function pointers are used to decouple the component connections in a similar manner to C++ vtables. Lorien uses a slightly augmented C to write programs, however there is little discussion of a programming model, particularly in relation to the concurrency model. Developers are require to manually specify the dependences between components, and the system does not support runtime discovery of entities or location transparent communication.

As previously mentioned, the default method of reprogramming in TinyOS is image-based, however, **Dynamic TinyOS** offers module-based replacement. Munawar et al. [43] have modified the compilation process of TinyOS so that a monolithic binary file is no longer created. The user specifies one or more nesC components to be included within a module, and the compiler then generates a corresponding ELF file for the module, just as in Contiki. The general problem with this approach it that TinyOS is designed to execute with all information available at compiletime, including the number of *tasks* to allocate a large enough array for the scheduler, or the number of required timers. To mitigate these issues, such resources are over-provisioned to safeguard against these problems in the future. This requires the allocation of resources in a pessimistic way. The advantage of this approach is that TinyOS application elements can be replaced, rather than replacing the entire OS image. However, it is not clear how flexible this replacement strategy is, for example, if the user initially selects multiple components to constitute a module, is it possible to separate these components at a later date into separate modules? Migration, runtime discovery of entities, and location transparent communication is not supported.

AFME [44] is a framework which uses the declarative Agent Factory Agent Programming Language (AFAPL). The entire system is implemented in Java for the sunSPOT platform [45]; the sunSPOT is a sensor mote specifically designed for Java. The language itself does not provide state encapsulation between agents. In order to enforce state encapsulation, all communication must pass explicitly through well defined software modules, however this is not enforced by the system. Inter-node communication is possible, however this is explicit, with the user being required to enforce reliability, if desired. In terms of migration, the system can only support *weak* migration (Section 2.2.5), as the JVM prevents a Java thread access to its internal state. Also, the system does not deal with *object-entanglement* - i.e. should an object reference a co-located object, a copy of the object must already exist at the remote location. Remote references, runtime discovery, and remote object loading are not supported.

Middleware

LooCi [46] is a component-based middleware platform for WSNs, which was initially built on Java to work with the sunSPOT platform, and subsequently Contiki and OSGi [47]. Unlike the channels found in this work, LooCi uses a global event bus to enable communication between components. A component registers interest in certain events, and advertises the events that it generates. It is the responsibility of "intelligence in the networking layer" to ensure that events are delivered appropriately, thus abstracting over the underlying communication medium, however, this intelligence is never explained. LooCi supports the relocation of components using the mechanisms of the underlying system. For example, Contiki macros are added to create components and events, and the Contiki elfloader is used to move the components. Again, this offers replacements and relocation, but not migration. It is possible to store an ELF file on a mote for later dissemination, however the general model requires motes to query a *component store* located on a PC behind the WSN gateway; location transparent communication and general runtime discovery of entities is not supported. Currently, components can not be moved between systems; a Contiki component can not be moved to a Java system. As LooCi is a middleware platform it can take advantage of existing systems, however this means that there is no unifying programming model.

2.2.2 Virtual Machine Replacement

A Virtual Machine (VM) is a software implementation of a machine that executes a custom set of instructions often known as bytecodes. The most popular example is the Java VM. The advantage of VMs is that they present a homogeneous abstraction layer, regardless of the underlying hardware. Consequently, an application can be compiled into bytecodes and then executed on any hardware platform that has an appropriate virtual machine. Furthermore, the single hardware abstraction presented by VMs simplifies runtime adaptation. On resource-rich hardware, such as desktop PCs, the use of VMs is not an issue; however, due to the resource-constrained nature of sensor motes, the implementation and use of a VM is challenging. For sensor networks, the advantage comes from reduced radio power usage as bytecodes are smaller than binaries, but at the expense of runtime interpretation.

Maté [48] is an application-specific virtual machine which executes as a component on top of TinyOS and interprets Maté scripts. A Maté application is a collection of up to eight *capsules*, each containing 24 assembly-like instructions. The interpreter itself consists of an operand stack and a return address stack. Maté was the first VM on a mote. Due to the size restrictions placed on a Maté application, the type, size, and complexity of applications which can be written is severely restricted.

More recently, work has been done to create a Java compatible VM on sensor motes, specifically **Darjeeling** [49] and **Mote Runner** [50]. Both systems implement a 16-bit stack operand-sized instruction set and support a subset of the Java bytecodes, and by extension the Java language itself. The most notable omissions are reflection, arraylists, generics, and recursion. Both approaches use the *split-vm* approach, where class files are linked on a desktop machine into a much smaller and denser format. The VM running on the node executes this format rather than the original class files.

There are two distinct differences between these systems. Firstly, Darjeeling is designed to support the Java language, where as Mote Runner is designed to support multiple high-level languages including C#, Java, and JavaScript. The goal of Mote Runner is to remove the learning curve required for traditional WSN programming. Ultimately both techniques are compiled to bytecodes, however Mote Runner has the more challenging task of unifying a number of different programming abstractions to be applicable on sensors, as well as injecting constraints (as discussed below) into these high-level languages.

The second difference is in the concurrency model provided. Darjeeling offers stack-based threads which are dynamically resizeable, whereas Mote Runner offers event-driven concurrency via callbacks, as used in nesC. Mote Runner believes that stack-based threads are not suitable in WSNs. As a result, Mote Runner application logic must be specified in terms of function callbacks.

A more general point about VMs is performance when compared to native code. The cost of portability is that time must be spent interpreting the bytecodes. For sensor networks this cost is shown in detail by Dunkels et al. [38] and highlights that VMs take longer to execute and consume more power than native code. The advantage is that it provides a convenient platform to explore runtime adaptation.

2.2.3 Non-WSN Computing

Xen [51] is a virtual machine monitor (or hypervisor) which allows a number of operating systems to run on a single machine simultaneously; Xen itself is the only element to run on the actual hardware. The other operating systems run within virtual machines known as domains. Xen manages each domain's access to the physical resources and prevents different domains from interfering with each other - i.e. by two nodes trying to concurrently access and modify the same area on disk.

Xen uses a virtualisation technique known as paravirtualisation. This is contrast to full virtualisation as used by VMware [52]. No changes are required in a system to work with VMware, whereas to work with Xen an operating system must be ported in a similar fashion as a system is ported to a new piece of hardware. While this requires more work for the developer, the advantage comes from better runtime performance.

Within Xen it is possible to pause, save, move, and replay a XEN domain from one machine to another. Further work has been done to show that this can be done at runtime with very small overhead and downtime [53]. This work shows that it is possible to not only migrate domains as a proof of concept, but also in real world examples. This approach is designed specifically for the administration of clusters of computers. As adaptation is at the OS level, it is a heavy-weight and coarse-grained approach.

Mirage [54] proposes an alternative application of Xen to large-scale, distributed computing, sometimes referred to as *cloud* computing. Mirage is a programming framework that enables a user to write an application in a dialect of the Objective Caml language. Each application is then compiled into a custom, standalone Xen domain. This has the advantage of removing the OS and many layers of software required in a traditional Xen domain, leaving an application-specific binary. This is in the same spirit as conditional compilation.

In terms of performance, testing against custom database benchmarks have shown that Mirage performs better than an equivalent application running on Linux at scale. However, certain details are missing from the description of the testing. Specifically, it is claimed that Mirage performs better as the scale of the application grows, however there is no discussion of the number of instances used or how these are allocated. The minimum binary size of a Mirage instance was 600KB, two orders of magnitude smaller than the Linux equivalent and several orders less than the Windows equivalent, however still at least one order of magnitude too big for a sensor mote.

The work of Giurgiu et al. [55] explores the migration of sections of an application between mobile devices and the cloud based on using Java and OSGI component modules. The work addresses the issue that static partitioning of applications for code offloading in mobile phones is not adequate. Instead, they have implemented a system which dynamically profiles an application, and decides what and when to offload code to the cloud. The system is built upon R-OSGI [56]. As applications in this system are written in Java, there is no linguistic mechanism to enforce loosely coupled applications, hence the developer must be relied upon to create suitably partitioned code.

MPI [57] is a well-known framework for message passing communication which is supported by a number of different programming languages, and has been used by many systems, particularly in high performance computing[8]. The system is accessed via a per-language API. In MPI different loci of execution are known as processes, where processes are assigned to CPU cores. These processes can be spawned across local or remote machines, where each processes is uniquely identified by its rank. The total number of processes and their ranks are determined when an application is launched with MPI.

MPI supports remote creation of processes and location transparent communication, although users must manually marshal/demarshal complex data types in the language. Furthermore, there is no compiletime support for type checking two ends of the communication pipeline, although session types can be used to help address this issue, Section 6.3.1. MPI supports static hardware discovery based on predefined configuration files, however, is it not designed for a dynamic execution environment, where nodes come and go. MPI does not support transparent process migration. If desired, users must use a manual checkpoint and restart mechanism at the language level. Finally, due to the API-based nature of MPI, it leads to low-level, verbose applications [58]. While this gives fine-grained control over application development, it can act as a barrier to non-expert programmers.

An alternative to Java and the JVM is **Forth** [59, 60]. Forth is a language and runtime which

[8]http://www.open-mpi.org/ - Accessed May 2015

is composed of *words*, symbols, such as “+” or “-”, and numbers. These words can either be *well known* or defined by the developer. Words are kept in a *dictionary* which is consulted at runtime to find the definition of a word. A program or word is expressed in reverse polish notation and consists of words, symbols and numbers. Forth interpreters are very simple, small, and easily extensible as most of the work is done in specifying the words for the dictionary. Forth code can either be interpreted or compiled. Different interpreters deal with missing words in different ways, although there is no clear strategy. Some will store missing words to be populated later, some throw errors. One advantage that Forth has over the JVM is that new words can easily be added to the dictionary at runtime, thus new functionality is very easily available. By contrast, the Java VM itself would need to be modified, recompiled and reinstalled in order to add a new bytecode.

In a similar ethos, both **Python** [61] and **Ruby**[9] support dynamic code generation and execution at runtime, although this is done at a comparatively higher level within the programming language. While these would support the remote creation of code instances quite easily, there are a number of draw backs. The runtimes for these systems are comparatively large, requiring 13.1 MB and 6.1 MB for Python (3.2 minimal) and Ruby (1.9.1), respectively, to support a hello world application. These sizes are larger that the storage capability of some of the embedded devices targeted in this work. This is not to say that a smaller runtime could not be created, but a lower level intermediate representation offers a simpler runtime, requires less data during remote communications, and potentially provides more scope for compile-time/runtime optimisations.

Like the JVM, the common language runtime (**CLR**) [62] is a virtual machine providing services such as memory management, security, and exception handling, and is also designed to execute a common intermediate language. The CLR is a part of the .NET framework. Unlike the JVM, the CLR was designed to execute multiple different languages from the outset, whereas the JVM has become the target of many different languages [63]. **Mono**[10] is an open source version of the .NET framework.

Given the existing support for Java, a subset of the Java bytecodes were chosen as the starting point for the Intermediate Representation (IR) of the language. These bytecodes were then extend, and a custom VM was implemented across the scale space. The modifications made to the bytecodes and the custom VM are discussed in Chapter 4.

2.2.4 Service Discovery

Given the nature of the distributed systems being targeted by this work, it is not realistic to rely on persistent connections between physical devices. This requires the use of an on-

[9]https://www.ruby-lang.org - Accessed May 2015
[10]http://www.mono-project.com/ - Accessed May 2015

demand discovery mechanism to locate entities which are currently visible. There are a number of approaches for providing such a service. Common amongst them is the need to associate some set of properties with the entities which can be discovered at runtime, and the ability to query those properties to gain references to the entities they represent. This approach is used in Jini [64], CORBA [65], dns-ds [66], and Bonjour [67]. The design of these systems has influenced how the location of actors and stages discussed in Section 3.4 is expressed in the language and implemented in the runtime. The following expands on two fundamental points raised by these technologies.

Push vs. Pull Discovery

When locating entities at runtime, the publication of properties associated with an entity can either be recorded locally, requiring a query operation to search remotely for these entities, or the publication can push this information to all other remote sites.

Having publishing devices push information to other devices reduces the time and network traffic for query operations, but requires greater amounts of local storage space at each remote site, and can lead to remote sites hosting information which is never used or out of date. Conversely, having the query search remotely causes more network traffic, but saves on local storage space. As queries only occur when required, only currently accessible devices will be interrogated, and any information is less likely to be out of date.

Structural vs Named Equivalence

The ability to determine if two types are equivalent is necessary both statically at compile-time and dynamically at runtime. The equality of two types may be determined via a type's name, or a type's structure [68].

Name equivalence means that types with the same name are considered equivalent. Note that these names are often compiler generated, rather than the type names used by the programmer. Here, two types are equivalent if they have the same name. The advantage of this approach is its simplicity. The disadvantage is that applications compiled independently may have equivalent types with different names. Two types which are the same but with different names will not be compatible, even if this is desired. Name equivalence is often considered a very restrictive approach.

Structural equivalence means that two types with different names are considered equivalent, if for each feature within the second element's type, there is a corresponding and identical feature in the first element's type. By ensuring that two types are equivalent based on their structure, functional correctness is guaranteed because the compiler can ensure that the correct operations are applied to the data types, even between code which is compiled independently. The disadvantage of structural equivalence is that two equivalent types may have different associated semantics. For example, even though a *child* type and an *adult* type are structurally equivalent, an *adult* type may be used in a very different manner. If only using structural equivalence, a *child* type may be used incorrectly in the place of an *adult*. Structural matching is often considered a liberal approach to equivalence.

2.2.5 Offloading Techniques

In the 1980's through to the late 1990's, much work was done on **process migration** in desktop computing. Milojicic et al. [69] have surveyed this field and give an overview of the application areas, research and implementations, and present reasons as to why migration never caught on.

In general, the application areas for process migration cover load balancing and distribution, the exploitation of resource locality, resource sharing, fault resilience, systems administration, and mobile computing. These concepts can be summarised by saying that process migration enables applications, either autonomously or with outside direction, to be liberated from their current execution environment for the purposes of efficiency, safety or policy. The survey notes suitable applications areas are those that exhibit parallelism, long lifetimes, and mobility; this is corroborated by Smith et al. [70].

The aforementioned research on process migration covered many different areas and many different systems. The general findings can be summarised in a number of points.

Firstly, migration-capable systems which use message passing are easier to design and implement than those that do not, especially in terms of location transparent communication. This is because message passing offers a decoupling of interaction between software entities. This opinion is also expressed by Smith et al. [71]. However, it is noted that this advantage is at the expense of complex communication logic within the runtime. In order to ensure message delivery after process migration, some approaches use message forwarding; however, this can lead to complex network routing and is not efficient. A more desirable approach is taken by the V kernel [72] (and Xen), where routing information is updated after migration, however the details are not discussed in depth.

Secondly, environmental entanglement is not solved, with all application and user level approaches requiring that any process to be migrated must be isolated. Some of the micro kernel approaches, particularly Mach [73], provided distributed shared memory and distributed inter-process communication to mitigate some elements of entanglement, although this can lead to very complex runtimes.

Thirdly, one of the factors which contributed to the lack of success for process migration was complexity within the runtime systems. This is especially true for Mach, where the support for distributed memory management and IPC became very complex.

Other examples included a lack of applications, migration not being required as other technologies (RPC, discussed below) were available and well understood, a lack of wide spread infrastructure support - i.e., Windows did not support migration, and security concerns.

The work does suggest ways to address these issues, however these are quite general and based on assumptions about the computing environment and changing trends in computing technology, some of which have only now been realised in terms of mobile computing hardware, and the applications that are executed on them.

Other work on migration focused around modifications to Java to extend the serialisable functionality which was already present; to not only move a component from one location to another, but to enable it to also take its execution state. Generally this work fell into two categories, one which modified the Java VM to support migration [74, 75], and preprocessing of Java code to insert mechanisms to save the state before migration and restore the state after migration [76], the so-called *checkpoint-restart* approach. The work of Baumann et al. [77] describes another system which provides mobility around Java, but also presents a classification of different types of mobile programming by introducing different levels of mobility: *remote execution*, *weak migration*, and *strong migration*.

Remote execution covers remote produce calls (RPC) and the Java equivalent, remote method invocation (RMI) [78]. This is the ability to invoke a procedure or method on a remote machine. Here a *stub* piece of code is automatically generated during the compilation process on each machine. Its duty is to abstract over the heterogeneity of different machines as well as handle networking issues such as marshalling. It should be noted that RPC/RMI operates within the semantic space of a single memory region, as opposed to actors which use the shared-nothing semantics to create applications with distinct memory regions. Hence, actors are more naturally suited to distributed applications.

Weak migration is the ability to relocate the code and data but not the process state. Considering an actor system, this means that an actor can be inserted at a remote location, but will start from the beginning of its behaviour. This provides the ability to replace, and relocate an actor. Migration is still possible with this scheme, but requires the actor's behaviour itself to orchestrate this, using the checkpoint-restart method.

Strong migration refers to the ability to relocate the code, data, and state of a process. Again, considering an actor system, this is the ability to pause an executing actor, save its state, transport it to another machine, and let the actor resume execution as though nothing had happened.

2.3 Accelerator-Based Programming Approaches

To address the limitations of single core CPUs, accelerators, such as multicore CPUs and GPUs, and co-processors, such as the Xeon-Phi [79], provide the user with multiple physical threads of execution, thus enabling many computations to occur simultaneously. As described in Section 3.3.1, programming such devices is primarily achieved by having some controller logic (host) situated on the main CPU which will initialise, control, and communicate with some application logic (kernel) on the accelerator. This approach closely relates to the actor model of computation, where different loci of computation communicate explicitly. This is discussed further in Section 3.3, but to provide context, the following presents a survey of the state of the art in this field.

A number of different approaches have been designed to program parallel computing hardware platforms and are discussed in the rest of this section. These styles can be categorised into three equivalence classes: API access, semi-automated parallelisation, and automated parallelisation.

2.3.1 API Approach

The most low level approach to programming an accelerator is to enable access via an API within an existing language. Examples of this include Python [80] , Java [11], and the original implementations in C/C++ of OpenCL [81] and CUDA [82].

While an API is a simple approach, requiring no modifications to the host language, the general drawback of using an API is the need to write large amounts of boilerplate code simply to setup the programming environment, as described in Section 3.3.1. This boilerplate code often follows the steps required to setup and initialise the relevant framework, and can have very little relation to the programming idioms of the host language - e.g., the Java API requires the use of pointer objects. Similarly, the code required to express the calculations on the accelerator is written in a C-like language, which is embedded in the host language as a string. For non-C-like languages, such as Java and Python, this can be challenging for programmers without experience in C/C++. More generally, this leads to two different programming styles for the host and the accelerator.

2.3.2 Semi-automated Approach

Much work has been done on semi-automated translation of serial programs to parallel OpenCL or CUDA code. OpenCL is discussed further in Section 3.3.1. The techniques used

[11]http://www.jocl.org/ - Accessed June 2014

include recognition of common parallelisable patterns in the source code, such as nested loops that can be unrolled and executed in parallel. However, most rely on the programmer to provide annotations or some form of refactoring applied to the original code.

OpenACC [83] is a set of explicit annotations for C or Fortran code. The simpler annotations are similar to macros, and are used for allocating and writing to buffers. However, higher level annotations also exist, which offer substantial control over how a loop should be unrolled and scheduled. Using these hints, the compiler attempts to generate well-optimised code for the supported patterns. The system allows the use of previous code written in the original language with fewer modifications than is required to use the OpenCL API. OpenMP [84] is also a directive-based approach which targets single/multicore CPUs, but is supported by open source compilers, unlike OpenACC.

By their nature, annotations are extraneous to the logic of the underlying application. This has the advantage that the existing logic can be used as a starting point, with annotations extending or enhancing the functionality. However, annotations must be applied to *each* construct to be parallelised, resulting in applications which are increasingly difficult to follow as their size increases. This is also true for any library code. Also, there is no guarantee that the compiler will be able to generate an effective parallel strategy for the annotated section of code. For example, if there is a non-linear data dependency in a for loop, sequential code may be generated instead of parallel.

hiCuda [85] is a similar approach for the CUDA framework, applying annotations to sequential C. The paper indicates that annotated applications provide similar performance to hand-crafted CUDA.

AMD's open-source **Aparapi**[12] system allows partial offloading of Java code at runtime, depending on the available OpenCL device's capabilities. The same program may execute on a system without OpenCL support, where it will use a Java threadpool. Aparapi relies on the programmer to refactor a function or loop into an inner class with a run method containing the computation. The Java bytecode for this method is translated to OpenCL/C code at runtime. The programming model is similar to this work as it requires some refactoring but still allows the kernel code to be written in a subset of the original language, making use of the primitives (such as classes) provided by that language to express the required meta-information. It also abstracts, to a certain extent, the exact memory layout and management of data movement and provides automatic adjustment to the available devices, rather than having the programmer optimize the code for a specific device. Aparapi offers a trade-off between a simplified programming model, and a non-trivial performance cost [86]. A much smaller performance penalty is found when using Ensemble, Section 5.2.

Functional languages, such as Haskell, provide an alternate approach. Purely functional

[12]code.google.com/p/aparapi - Accessed October 2014

languages are side-effect free. In principle, this enables the runtime system to extract parallelism by executing expressions in parallel. In practice, it is difficult for the runtime system to ensure that a given expression is large enough to warrant the overhead of forking a thread to compute its value in parallel with other expressions. Glasgow Parallel Haskell (GPH) [87], therefore, provides a *par* annotation, which programmers can use to identify promising expressions for parallel computation. The par annotation does not change the semantics of the program, instead enables exploitation of the existing potential parallelism. This is similar to the directive-based approach, but much simpler due to the comparatively constrained programming model of functional programming. Concurrent Haskell [88] provides additional operators to enable threads to be forked explicitly, thus enabling a developer to express a program in a concurrent manner, if this is appropriate. However, exploiting the potential parallelism provided by both concurrent and parallel Haskell efficiently continues to be a challenging problem [89].

2.3.3 Automated Approach

A different approach to simplify the creation of massively parallel programs is to hide from the developer all the low-level details such as memory allocation, the specification of work sizes, and when to dispatch a kernel. A number of new domain-specific languages have been developed to provide such higher level abstractions. Their primitives describe data movement and computation through operations such as map, reduce, stencil, and other vector operations. By separating the description of the algorithm from the implementation, the underlying OpenCL/CUDA code generation can be swapped out transparently to the user, and, at least in theory, pick the best representation for the available device. In general, the effectiveness of the approach relies on the quality of the code transformation from source language to OpenCL/CUDA.

One of the earliest efforts in this direction is the **Accelerator** system developed by Microsoft Research [90]. This extends the C# programming language with lazy and immutable parallel arrays, which can be converted to and from normal arrays. These parallel arrays can only be operated on as a whole by using a large set of predefined operations. The runtime system Just In Time (JIT) compiles DirectX shaders, and performs the accumulated operations when the arrays are converted back. Accelerator pre-dates both OpenCL and CUDA, and is an early approach to simplifying general purpose use of GPUs. This approach could be extended to generate OpenCL kernels for the array operations instead. The runtime system can optimise the shader code for the available device architecture, and the authors report performance similar to hand-optimized shader code, with the main hindrance being the JIT compilation step and resolving the dependency graph at runtime.

LIME (Liquid Metal) [91], developed by IBM, is a Java-based language with added parallel operations. Its semantics are based on task and connect statements, as well as explicit map and reduce operators. Tasks are functions that can be connected together at compiletime, enabling static topologies. This enables the compiler to effectively optimise code, however making it impossible to change inter-task topologies at runtime. Such dynamic reconfiguration is useful for load balancing or reprogramming at runtime. The optimising compiler detects when such a function can be parallelised, for example by checking that it has no side-effects. (The Ensemble system also has no global side-effects, but does not require immutable state.) The LIME compiler decides which functions to offload, and scans for data parallelism in functions that behave as filters. A number of heuristics for optimizing the use of OpenCL memory regions are demonstrated. Marshalling is used to transfer data from the Java program to native code (implemented via the JNI). It is de-marshalled on the host in native C code, before being transferred to the OpenCL device. The authors admit that the primitive implementation of this marshalling is relatively slow. It is not clear from this work how kernel work and group sizes are determined, though it can be assumed to be based on the data size. LIME compiles from augmented Java to a program composed of Java with native (JNI) code and OpenCL/C kernels.

SkelCL [92] is a library of hand-implemented and optimized skeletons of parallel programming patterns for computation and communication which are then filled in with user functions. It targets OpenCL systems, including those with multiple GPUs [93]. These skeletons are optimized to avoid memory bank conflicts and similar issues. User functions are passed as source code strings and merged with the skeleton code before being compiled to OpenCL kernels. In effect, they work like higher order functions. One drawback of this approach is the need to use stack-like operations to assign variables and values to multi-argument functions. Also, as the compilation of these functions happens at runtime, the user must wait before compiler errors or warnings are displayed. Lazy copying is employed to avoid data transfers if the next user of a piece of data is running on the same device. This is a feature that the Ensemble language expresses explicitly through *movability* within the type system, discussed in Section 4.5.2. As SkelCL hides the use of OpenCL, the different memory regions are not accessible to the programmer. While this does simplify the programming model, it limits the use of the more powerful features of OpenCL which are often required for non-trivial applications. The authors show performance results that are comparable to handwritten CUDA and OpenCL code, with fewer lines of code.

Chestnut [94] is a data parallel language built around special types, such as parallel arrays and vectors, that are used to define the parallelism of loops using these data structures. It is especially suited for stencil and grid operations, providing methods to easily access adjacent elements in a multi-dimensional array within a parallelised loop. Chestnut is compiled into C++ with the author's own *Walnut* library which includes CUDA code to access the GPU. The evaluation shows performance close to hand-written CUDA code. The authors also introduce a graphical designer that can be used to generate Chestnut source code, which can then be modified to the programmer's needs. Another novel feature is a built-in visualisation method to show the computation results. The language does not support structs or objects yet, and is limited in the types of parallelism that can be expressed.

2.4 Program Analysis

One of the issues associated with shared-nothing semantics in the actor-model is the cost in terms of runtime resources, as described in Section 3.2. This manifests itself as the need to duplicate data when it is communicated between actors. The use of language types and compiletime analysis can be used to mitigate these costs; this work describes these as *movable types*. To place this work in context, the following is a summary of related efforts.

2.4.1 Movable Types

The ideas expressed in this work are similar to other concepts within the literature. If data transmission between actors is considered as an operation on the type of memory being transmitted, then modifications can be made to this type to dictate the number of times that data can be sent; if the data can only be sent a single time, then there is no need to duplicate the data.

The Islands approach [95] describes the use of *bridges* to encapsulate *islands* of state within object-based languages in the presence of aliasing. Conceptually, an actor is a stricter form of a bridge, since the state inside an actor will never escape its scope. Destructive reads are similar to the movability property. The approach in this work differs in syntactic cost; the compiler tracks movable types throughout the program from a single annotation at the memory allocation point (**new**), whereas (in addition to object allocation points) the "access mode" of parameters and function results must be specified in [95].

Rust [96] has a similar memory and concurrency model to this work. A *task* in Rust is similar to an actor with defined communication channels, although Rust uses static channels defined at compiletime. In Rust, communication is performed using *pipes*; a *channel* is a sending endpoint of a pipe, a *port* is a receiving endpoint for a pipe. The notion of channels and pipes are equivalent to `in` and `out` channels in Ensemble, respectively. Tasks cannot share data with each other and must transfer ownership using a global *exchange heap*. The *Send trait* acts to communicate data between tasks, ensuring that the data is no longer used by the sender after being sent. The Send trait allows only *owned* boxes to be sent between tasks, where an owned box is an object that has a single pointer (owning pointer) to it. A *managed* box is an object that can have any number of pointers (managed pointers) to it. The heap can be viewed as split into regions for owned and managed boxes, respectively.

The notion of *borrowing* is provided in Rust by *borrowed pointers*, to which managed or owning pointers can be assigned using an automatic pointer conversion operation provided by the language. For example, an owning pointer can be borrowed by passing it as an argument to a function accepting a borrowed pointer. During the execution of the function (known as the *lifetime* of the borrowed pointer), the owning pointer cannot be used since the object is on *loan* to the function; the function is known as the *borrower*. When the borrower returns, the owning pointer may be used again. Further, a borrower cannot send the object over a communication channel to another task; the object may only be sent from an owning pointer. These rules for move semantics in Rust require the programmer to know about lifetimes and pointer conversion operations. While Rust provides a rich set of semantics for data movement, a non-specialist programmer will find it difficult to grasp such concepts easily.

This work has been strongly influenced by Rust, and the movement property is an amalgamation of the managed and owned pointers in Rust, allowing multiple references to the object within an actor (analogous to multiple references within a task), yet still maintains the task-level ownership analogous to an owned pointer in Rust. Additionally, there is no need for the programmer to understand lifetimes or pointers since these concepts are not made explicit in the move semantics defined in Ensemble (see Section 3.2.2). Instead, the programmer simply annotates heap allocation points with `mov` and the compiler tracks references to all such objects. No other annotations are required at any point as this is tracked entirely by the compiler analysis (see Section 3.2.4).

Ownership types [97] were developed for providing strict, static object encapsulation for object-based languages. Clarke et al. [97] describe two type annotations to define variables which are accessible globally (*norep*), and those which are only accessible by the object which created it (*rep*). The work described in this dissertation provides the *rep* semantics for all actors by definition, as an actor is the *owner* of its entire state. “Ownership transfer” is conceptually the same as the movability property, allowing objects to “jump” across the *articulation point* represented by the actor. In the object graph described by Clarke et al. there is no dual of the channel mechanism, however, this can be considered as gateways or bridges (as in [95]) to allow an object to move across the boundary. It should also be noted that alias protection is reserved with movability since only one actor will have any aliases to an object at one time, though that actor can have any number of aliases of its own.

Uniqueness types [98] are similar to ownership types, but apply at the variable level, ensuring that a variable is used in a single threaded way. This approach requires that there only be a single reference to a variable which has been declared `unique`. Once a variable of this type has been used, in a function call for example, the type system ensures that it cannot be used again after the function returns. However, a new distinct variable of type unique can be created which points at this data and is returned to the function caller, thus preserving referential transparency. The state encapsulation presented by actors ensures that messages sent between actors are unique, while not being the case within actors internally. Alternatively, movability enforces uniqueness on movable types which are sent to different actors.

A type system that allows transference of object ownership by using a “permission” based mechanism is presented by Naden et al. [99]. The system provides a simple mechanism for procedures to borrow, or even consume, an object through changing the permission attribute. The system could easily provide higher-level abstractions on top of the defined permission semantics to provide a form of the movability property. Indeed, a movable type acts in much the same way as an object initially set with a “shared” permission. Send and receive primitives could be defined as functions which change the permission of the provided object to “none”. The system also gives the programmer tight control of aliasing through the permission system. The programmer has to explicitly manage the permissions, and have detailed knowledge of language theory to understand the effect of permissions on aliasing, which could detract from the task of developing the application. The approach in this work minimises the burden placed on the programmer, requiring very few changes to an application to enable the extension, using compiletime analysis to perform the movability tracking throughout the program and the management of aliases.

2.5 Summary

In order to program recent generations of hardware devices, a developer must be familiar with thread-based, accelerator-based, embedded, and distributed programming. Additionally, they must also be aware of the different hardware constraints of the potential hardware platforms that they may use. Given these hardware platforms and the different operating conditions that they present, there is a growing need to modify an application or its execution environment at runtime. This places an additional or unrealistic burden on the developer, who is often a non-expert, and increasingly a non-computer scientist.

The previous discussion in this chapter has given an overview of the state of the art in this area of programming. These approaches are highly tuned to their specific problem areas. A number of different actor languages have been surveyed in Section 2.1. The languages discussed either focus on a single problem area, or are in some way disadvantaged, either because of the necessity to implement system functionality at the language level, constraints imposed by the language itself, or a non-strict compiler which enables inappropriate interaction with the underlying language. Section 2.2 surveys a number of different techniques required to perform program adaptation, including discovery and reconfiguration, at different levels of computing scale. While they exist, they are not integrated with programming models, do not support strong migration, and are not supported across multiple scales of heterogeneous hardware platforms. Section 2.3 describes three approaches to programming parallel hardware devices. These integrate easily with existing programming styles at the cost of brevity, require new programming styles and increased performance costs, or increase existing program complexity and obfuscate the flow of logic. Finally, Section 2.4 provides an overview of program analysis.

So far, there has been little work towards providing a single programming abstraction for all of these challenges or supporting this across a wide range of heterogeneous platforms, such that non-specialist programmers can easily exploit parallel or distributed heterogeneous systems.

Chapter 3

The Ensemble Programming Language

The two prevailing trends of modern computing hardware are increasing interaction between heterogeneous platforms, and that these platforms are becoming increasingly concurrent. This new execution environment is challenging for the shared memory, sequential programming models which are currently used. Also, given the number and range of hardware devices, there are many different programming models and styles. As computing hardware is increasingly used by non-computing scientists who want to apply this new hardware technology to their own problem domains, existing programming approaches can be a limiting factor.

The goal of this dissertation is to show that an actor-based approach enables the creation of applications which take advantage of concurrent hardware and interconnected heterogeneous platforms. Additionally, this approach provides not only a functionally equivalent, more flexible and straight forward programming model than current models, but one that does not sacrifice performance.

This chapter discusses the linguistic approach to this goal, specifically the creation of a new programming language: **Ensemble**. The new language is based on the actor model of computation which consists of isolated loci of computation which interact by message passing. The point of this chapter it is not to show that the Ensemble programming language is the most appropriate actor-based language, but rather that it is a conduit through which to explore the concepts of actor programming.

The basic language constructs are described in the next Section, including a discussion of the communication semantics for actors, and abstraction of physical locations. A discussion of a *moveable* memory space which overcomes the limitation of the shared-nothing semantics actors and automated memory management is found in Section 3.2. Section 3.3 describes the integration of the OpenCL programming framework within the language which is the first use of actor as the abstraction of accelerator-based concurrency. The process of discovering and reconfiguring actors and stages at runtime from within the language is discussed in Section 3.4. The chapter then concludes with a summary of the points made.

3.1 Basic Language Structure

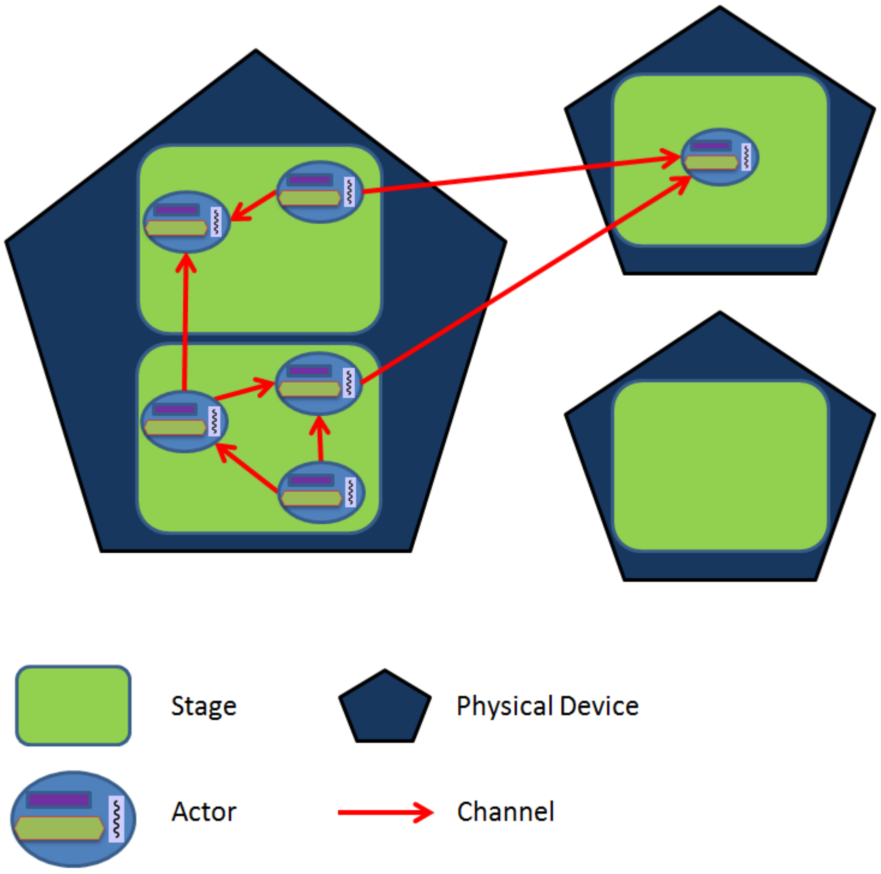


Figure 3.1: Ensemble Overall Architecture

Ensemble is a programming language based on the actor model of computation and the principles of the π-calculus [21]. It has been designed to simplify the expression of applications which are concurrent or distributed across multiple heterogeneous devices. The two core concepts of the actor programming model are message-passing communication and shared-nothing loci of computation. By imposing this *structure* on the design of applications, actor-based programming is implicitly suited to concurrent and distributed applications in a location transparent way. As there are a number of actor-based languages (Section 2.1), this section will only describe the salient aspects of Ensemble.

Figure 3.1 shows the hierarchical composition of the language. Applications expressed in Ensemble are collections of *actors* which interact via message passing along *channels*. Ac-

```
 1 type Isender is interface(out integer output)
 2 type Ireceiver is interface(in integer input)
 3
 4 stage home{
 5     actor sender presents Isender {
 6         value = 1;
 7         constructor() {}
 8         behaviour {
 9             send value on output;
10             value := value + 1;
11       }
12     }
13
14     actor receiver presents Ireceiver {
15         constructor() {}
16         behaviour {
17             receive data from input;
18             printString("\n␣received␣:␣");
19             printInt(data);
20       }
21     }
22     boot{
23         s = new sender();
24         r = new receiver();
25         connect s.output to r.input;
26     }
27 }
```

Listing 3.1: Simple Ensemble Send and Receive Example

tors are located within *stages*, which represent memory spaces; there may be one or more stages per physical device. Consequently, the main entities of Ensemble are: stages, actors and channels. Listing 3.1 shows a simple Ensemble application in which one actor sends a linearly increasing value to another across a connected channel within a single stage.

3.1.1 Actors

In Ensemble, an actor is a first class entity which represents a single locus of control with encapsulated state. An actor is defined with a name and one or more user defined interfaces (Section 3.1.3), as seen in Listing 3.1, lines 5 and 14. The interface is used to define the channels which may be used in the body of the actor. The following line references refer to Listing 3.1.

The body of an actor consists of three distinct sections. The first section is used to optionally

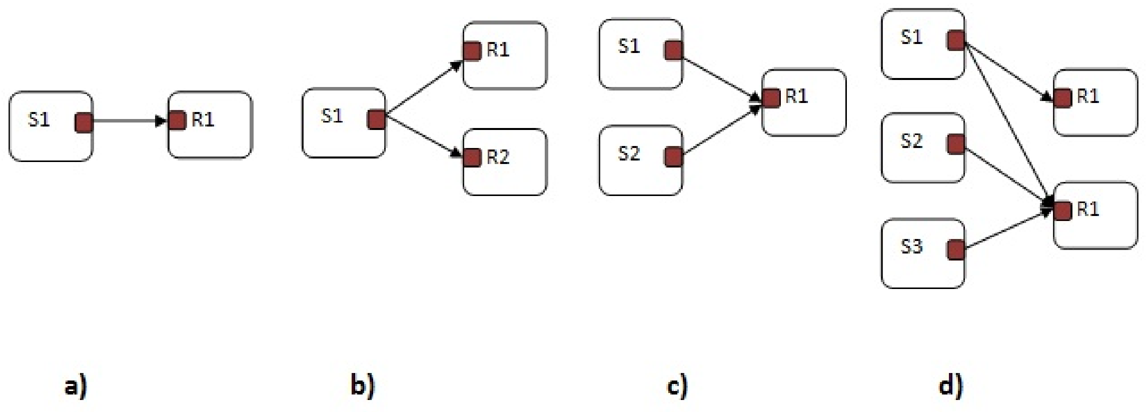


Figure 3.2: Ensemble Channel Configurations

define any actor specific state constructs (line 6). The entities defined in this section are only available within the enclosing actor. The second section is used to define one or more *constructor*s for the given actor (lines 7 & 15). Any actor-defined state may be referenced from a constructor. The third section is used to define the actor's *behaviour* clause (lines 8-11 & 16-20). The behaviour clause contains the logic of the actor, and is repeated infinitely, until explicitly told to `stop`. Telling an actor to `stop` will not kill it immediately, instead the actor will complete the current path through the behaviour clause before stopping. Killing an actor immediately would likely interfere with the overall application logic because of the channel connections, and the expected interactions via such channels, between actors. This choice simplifies the reasoning of stopping an actor.

Actors may create other actors, however actors may not `stop` other actors. To do so would violate the encapsulation of actors, and mitigate the simple flow of logic offered by per actor behaviour clauses. An actor must `stop` itself. However, one actor may send a message to another actor via the channel mechanism, requesting that the receiving actor commit suicide.

3.1.2 Channels

In Ensemble, communication between actors is achieved by passing messages along typed, unidirectional *channels*. Channels are first class entities in the language and are arranged into two sets: `in` channels which consume data, and `out` channels which produce data. This distinction is necessary as a channel represents one half of a connection; two channels must be *connected* together before data may be sent between actors. The two channels being connected must be of opposite direction (`in` + `out`), and convey the same data type. It is a compiletime error (and logically wrong) to connect two channels of the same direction or different types together. Channels convey data of any language type or user defined type, including channels. The ability to send channels between actors enables dynamic runtime topologies of actors. Figure 3.2 shows the possible configurations of channel connections at runtime.

Many actor languages choose to use implicit actor mailboxes to facilitate communication, hence messages are sent directly to actors. Ensemble uses multiple channels as it decouples actors from each other and enables the reconfiguration of channel connection topologies at runtime. Also, this enables channels to be used without concern for whether or not they are connected, as explained below.

Blocking-Rendezvous Channel Communication

By default, communication between actors in Ensemble follows the *blocking-rendezvous* model. This means that data is only passed between actors when both the sender and the receiver are explicitly trying to communicate.

For example, consider actor A trying to send a message on channel `output` which is connected to channel `input` in actor B. If actor A executes a `send` on `output`, the execution of the actor A will block until actor B executes a `receive` on channel `input`. Equally, if actor B executes a `receive` on channel `input` it will block until actor A executes a send on channel `output`. In this way, both actors must *rendezvous* before messages are sent.

By default, a deep-copy is made when sending data from one actor to another. This is done to enforce the shared-nothing semantics of the actor model, and ensures that each actor has a distinct copy of the data without requiring immutable state, hence, race conditions are not possible. When the data conveyed by a channel is another channel a duplicate of the channel, including any existing connections, is created and sent to the receiving actor. Once received, the channel is adopted by the receiving actor. This ensures that both sending and receiving actors have distinct channels, which have identical connections; hence this guarantees that one actor can not use a channel which is owned by another actor. Deep-copying in this manner is not always efficient, and is discussed further in Section 3.2.

There is no requirement in the language for a channel to be connected before a communication action is performed on it. In the previous example, should actor A attempt to `send` and actor B attempt to `receive` when their channels are not connected, they will both block indefinitely. Any subsequent `connect` operation on these channels will bind them together, facilitate the message being sent from A to B, and unblock A and B. This feature is useful in decoupling the overall application design and individual actor logic. Here the actor simply waits for input, without having to worry about the overall topology. This is common in distributed and event-driven applications, where logic will wait for input.

As well as single channel operations, it is also possible for an actor to receive data from multiple channels in a single action via the `select` statement, see lines 30-40 in Listing 3.2.

```
type Isender is interface(out integer output)
type Ireceiver is interface(in integer input)
type Iselector is interface(in integer input1;
                            in integer input2)
stage home{
  actor sender presents Isender{
    val = 0;
    constructor(integer init){
      val := init;
    }
    behaviour{
      send val on output;
      val := val + 1;
    }
  }

  actor receiver presents Ireceiver{
    constructor(){}
    behaviour{
      receive val from input;
      printInt(val);
      printString("\n␣was␣received");
    }
  }

  actor selector presents Iselector{
    example = 42;
    constructor(){}
    behaviour{
      select{
        receive val from input1 where example > 39: {
          // do stuff
        }
        receive val from input2 : {
          // do stuff
        }
        default : {
          // do other stuff
        }
      }
    }
  }
  boot{
    send1 = new sender(-100);
    send2 = new sender(0);
    send3 = new sender(100);

    recv = new receiver();
    connect send1.output to recv.input;

    sel = new selector();
    connect send2.output to sel.input1;
    connect send3.output to sel.input2;
  }
}
```

Listing 3.2: Ensemble Channel Interactions

A select statement is used to non-deterministically choose between one or more `in` channels based on the state of the channels and an optional boolean guard (`where` clause). The boolean guard values are used to decide which channels of the select statement may be chosen from, and forms a set of eligible channels. These expressions are evaluated at runtime, and may contain any literal, expression, variable or procedure which is in scope. Once a set of eligible channels have been determined, they are examined to determine if any are ready to provide data immediately. If there is more than one such channel, a non-deterministic choice is made and the data is retrieved from that channel with the same logic as `receive`. If there were one or more eligible channels, but none of which are ready to provide data, the actor will block on all eligible channels unless a `default` clause is provided, in which case the `default` clause will fire. A blocked actor will awaken when data is pushed to one of the channels upon which it has blocked.

The `select` statement is one way in which user level timeouts can be implemented. By selecting between a set of channels conveying useful data, as well as one registered with the timer actor (Section 3.1.8) to deliver data after a set time, the actor will only block as long as the timeout. Conceptually, `select` is similar to the select function in the sockets networking API [100].

Buffering

In practice, always blocking on single channels can lead to *channel* deadlock, where a number of actors are blocked awaiting messages from each other. This can be particularly problematic where messages are sent in non-deterministic order from actors which interact with the outside world; consider an actor waiting for a timeout. This is particularly true for actors which consume and produce data across channels at different rates.

For this reason, `in` channels may optionally be defined with buffers. Here, the communication semantics are modified as follows. An actor always sends on an `out` channel; a message will be buffered if the `in` channel to which it is bound has buffer space available. If there is no space in the buffer, the sending actor will block as before. When the receiving actor invokes a `receive` on the `in` channel, it will either retrieve any data in the buffer, or default to the blocking-rendezvous semantics. For the purpose of simplicity, only `in` channels may have buffers. The use of buffering enables asynchronous communication and hence provides a way to avoid deadlock. The correct provision of buffer sizes is an application specific detail.

It is important to note that the completion of a `send` operation guarantees that the message has been conveyed to an actor, but it does not guarantee that the actor has processed the message. If required, such guarantees should be implemented within the application. Also, communication guarantees ordering per connection, but not between actors. Consider the actors connected as shown in c) of Figure 3.2. In this case, all data sent from S1 will arrive in order relative to S1, and equally for S2. However, there is no causal ordering of data delivery between S1 and S2. Section 4.4.2 provides a discussion of how this guarantee is provided at runtime.

Another option to relieve channel deadlock would have been to perform analysis to determine if the current configuration of channel connections would result in deadlock, as is done in the GO language [101], however this was not feasible for a number of reasons. Firstly, in Ensemble, all channel connections are dynamic and occur at runtime. This precludes the use of static analysis at compiletime, as the topology is not static. Secondly, unlike GO, Ensemble's channels facilitate location transparency (Section 3.4), which would require distributed analysis of the connection graph. While this is possible, it is beyond the scope of this dissertation.

3.1.3 Interfaces

An interface is a language construct which is used to define one or more channels, Listing 3.2 lines 1 - 4. An interface is a static entity, consisting of only the channels expressed in its definition. Any channel defined in an interface is available within an actor which `presents` that interface.

In order to preserve the encapsulation semantics of the language, there are limitations on the actions possible on channels which are dereferenced from an interface. This is because such channels represent the actual channels of the actor referenced by the interface, rather than the channels of an actor which are connected to such channels. Consequently, channels dereferenced from interfaces may only be bound to, or sent along a channel. It is not possible to `disconnect`, `send` along, `receive` from, or `select` across such a channel. To do so would violate the isolation of the referenced actor, as one actor would be able to manipulate another actor's state directly. The `connect` operation is such a violation, however it is necessary to enable useful work between this actor's channels and the remote actor's channels. It is allowed as it does not modify any existing connections in the channel. In contrast, to perform a `disconnect` would potentially remove connections between the remote actor and third party actors, and is a clear violation of the semantics of the language. This does not prevent the local actor from performing a *disconnect* on the local channel which was bound to the interface channel, or having the actor referenced by the interface performing a disconnect on its local channels.

This discussion is mainly relevant to the use of adaptation, discussed in Section 3.4.3.

3.1.4 Stages

A stage represents a memory space within which actors operate. Conceptually, there may be many stages per physical computer, although currently there is only one per physical location; this is a limitation of the runtime, rather than the language or model, Section 4.4.2. A stage is defined with a name, which is used as an identifier at runtime, Listing 3.2 line 5. Note that this name is not necessarily universally unique, however there are other mechanisms which can be used to determine uniqueness, see Section 3.4. An actor may reference the current stage it occupies with the `here` keyword.

Within a stage there are two main sections. The first section is used to define language constructs to be used within this stage. This section usually contains actor definitions, but may also include query and procedure definitions, Listing 3.2 lines 6-42. Equally, this section may be left empty. Defining an *empty stage* is useful when creating an Ensemble environment across multiple physical locations for use with adaptation, Section 3.4.

The second section is the `boot` clause, Listing 3.2 lines 43-54. The `boot` clause acts as the *main* for this stage and will be invoked upon its creation. Any statement may be invoked within the boot clause, except actor specific actions, such as publishing properties (Section 3.4.1). Again, this clause may be left empty. The boot clause is primarily used to instantiate actors, and connect channels. Any actors instantiated within this boot clause will be created and execute within the context of this stage.

Whereas actors are entirely controlled from within the language, stages are controlled from the command line, outwith the language. This is discussed further in Section 3.4.6.

3.1.5 Types

Listing 3.3 shows the default types available in the language, as well as how to define new types. All types in Ensemble must be initialised with legal values when declared; there is no `NULL` type in the language. This is done to make the language safer and remove the possibility of invalid values. Although it is still possible to create incorrect code from a logical perspective, there will be no runtime type errors or illegal references.

Memory is allocated from the heap using the `new` keyword. As Ensemble uses automatic garbage collection to simplify memory management, there is no explicit action to return memory to the heap.

The Any Type

The `any` type is an infinite union type, which may be used to abstract any other type. Although a particular type may be cast to an `any` type, the original type can only be recovered

```
type Itype_examples is interface(out integer output)
type my_struct is struct(integer a)
type my_enum is enum(alpha, beta, gamma, delta)

stage home{
  actor type_examples presents Itype_examples {
    int_val     = 1;
    uint_val    = 1u;
    long_val    = 1L;
    double_val = 0.0;
    str_val     = "hello";
    bool_val    = true;
    struct_val = new my_struct(int_val);
    array_val  = new my_struct[100] of struct_val;
    constructor() {}
    behaviour {
      ...
    }
  }
  boot{
    s = new sender();
  }
}
```

Listing 3.3: Declaration of Ensemble Types

via the use of the `project` statement, as shown in Listing 3.4.

The project statement ensures that the `any` type can only be decoded to types which are defined within the current scope. This provides safety in the language as an `any` type cannot be arbitrarily cast to any other type. Should the underlying type of the `any` not be one of the types specified in the project clause, the mandatory `default` clause will be selected. In this case, the `any` type can still be used; for example, the value can be sent along a channel or used with another project statement.

One advantage of the `any` type is in combination with channels. By only receiving messages of the `any` type over a single channel, the receiving actor can either successfully decode the type and process the data, or discard the data. This enables logic more similar to traditional actor-based approaches which do not use explicit channels, but rather send messages to actors directly as proposed by Hewitt [4].

Collections

Currently, the only language-specified collection type is an array. Ensemble arrays are fixed length, although this size can be specified as either a literal value, or expression. Arrays may contain references but all elements must be of the same type. However, an array with elements of the `any` type is allowed.

```
type Iexample is interface(in any input)
type complicated_struct_type is struct(integer alpha;
                                       string beta)
...
behaviour{
  receive any_val from input;

  project any_val as mesg{
    integer : {
      printString("Got␣an␣Integer!\n");
      printInt(mesg);
    }
    complicated_struct_type : {
      printString("Got␣a␣complicated_struct_type!\n");
      printInt(mesg.alpha);
      printString(mesg.beta);
    }
    default : {
      printString("Don't␣recognise␣the␣type\n");
    }
  }
}
```

Listing 3.4: The Any Type and Project Statement

Given the types available in the language, higher level collections can be created. For example, it is possible to create user-defined linked lists using the `struct` and `any` types; linked lists are used within the draughts example, discussed in Section 5.3.1. Although it would have been useful to have a more complete set of collections, it was not the focus of this work.

3.1.6 Security

Security is not represented within the language as it was beyond the scope of the work. This said, Section 4.7 discusses security within the runtime.

3.1.7 Failure Model

The failure model in Ensemble uses a combination of explicit and implicit error handling. This is done to simplify the programming model, while enabling the option of fine grained control.

Explicit Error Handling

Explicit error handling is enabled via exceptions, which are defined by the language; it is not possible to create a user-defined exception. This choice was made to ensure that exceptions

Exception	Description
OutOfMemoryException	All heap memory allocated
NullPointerException	There was a NULL pointer found in the runtime
IndexOutOfBoundsException	Trying to access out with the bounds of an array
DivisionByZeroException	Trying to divide by zero
DuplicatePropertyException	Trying to publish an array of properties where there are two properties with the same key
StageNotFoundException	The supplied stage was not accessible
ChannelNotFoundException	The supplied channel was not accessible
ConnectionFailureException	A network error occurred
SpawnException	An error occurred with the spawn process
MigrationException	An error occurred with the migration process
ActorNotCompatibleException	The targeted actor may not be replaced by the one provided

Table 3.1: Description of Ensemble Exceptions

are used for truly exceptional events, as opposed to being used as flags. Using exceptions in this way minimises the complexity of code. The primary purpose of exceptions are to report on failures in the distributed features of Ensemble. Table 3.1 describes the possible exceptions in the language, and Listing 3.5 shows an example of explicit exception handling.

Implicit Error Handling

Implicit error handling is influenced by the *let it fail* model, as first introduced with Erlang [102]. In this model, programmers are advised to let their applications crash, rather than use large amounts of *defensive code* to protect against exceptional situations; in this model, it is better to let the application fail and propagate a failure message to all connected actors until one can deal with the failure message, or the entire system has died. At this point the system should restart.

The key advantage of this approach is that a crashed system will always restart in a defined state, as opposed to explicit exception handling which can lead to obfuscated logic flows; it should be noted that Erlang supports both approaches. This mindset is particularly relevant when applied to distributed systems, which are inherently unreliable. There is always a non-zero chance that unpredictable events will cause network failure[1].

Uncaught exceptions in Ensemble will always be implicitly caught and handled by the encompassing stage, at which point the stage will restart the actor. In this way, all actors are supervised by the stage on which they are performing.

A key difference between Ensemble and Erlang is that Erlang will propagate failure messages

[1]http://gizmodo.com/5644050/bored-hunters-in-oregon-are-regularly-shooting-down-googles-fibers - Accessed January 2015

```
actor calculate presents Iexample{
  // Some Initialisation

  behaviour{
    for i = 0 .. white1.length - 1 do {
      try{
        if(b.b[white1[i]] == 0 and ... ) then {
          score := score + mob;
        }
      }
      except IndexOutOfBoundsException {
        printString("check␣your␣bounds!\n");
      }
    }
    // Other work
  }
}
```

Listing 3.5: Exceptions in Ensemble

to all connected actors, whereas Ensemble will not. While this enables the failing actor and its connections to restart in a defined state, it limits the ability of Erlang applications to scale as a mesh connection must exist between all actors at runtime. This level of connection simply does not scale, and would be particularly difficult to support on the non-homogeneous networks types to which Ensemble is targeted. Consequently, Ensemble actors which fail do not propagate a failure message to connected actors. Section 3.4.5 discusses how failure is handeled with channels.

3.1.8 System Actors

There are a number of actors which are implicitly defined by the language. These actors exists to abstract direct interaction with hardware, and represent a way to avoid an application being tied to a specific physical location, also known as *environmental entanglement*. By using channels to interact with other actors, there is no difference in communication between application actors or system actors.

The use of system actors also provides a way to ignore the location transparency which channels normally provide, Section 3.4.5. For example, by having an actor which represents the sockets API, low level network programming can still be made available.

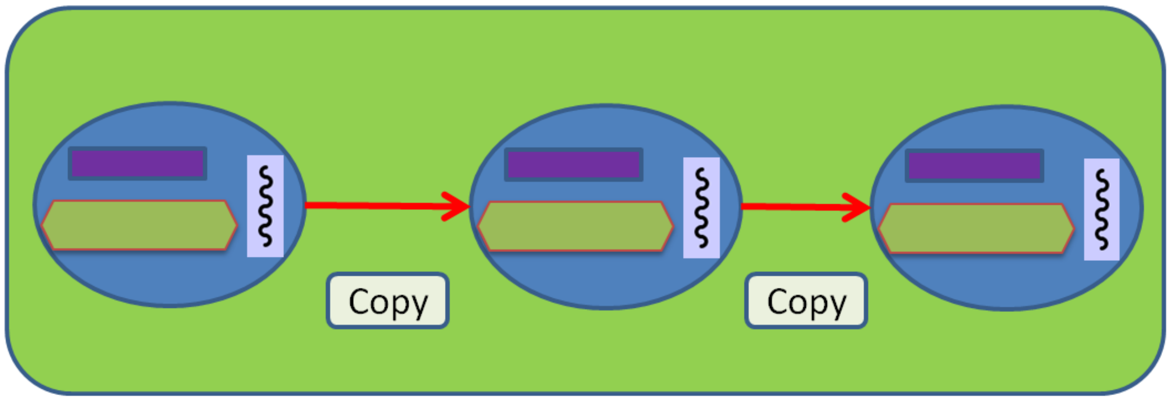


Figure 3.3: Duplication of Data for Pipelined Actors

3.2 The Movable Memory Space

The lack of shared state in the actor programming model makes it naturally suited to parallel and distributed applications. As there is no need for programmer-defined serialised sections of code, actors are inherently concurrent, without the concern of data race conditions. Message passing facilitates actor interaction, without sacrificing these advantages.

As described in Section 3.1.2, data is duplicated before being sent from one actor to another via a channel to preserve the shared-nothing semantics. While this does ensure correct semantic operation, it leads to increased heap usage and fragmentation. Furthermore, as Ensemble uses automated memory management it is not possible for the user to manually address this issue. This automation has the advantage that low level memory management is abstracted from the programmer, making the language safer and simpler, however, it also means that the user has no way to explicitly return a piece of memory to the heap when no longer required. Also, all data types in Ensemble must be initialised at creation to prevent variables being in an undefined state. While making the language safer, this means that there will always be data to deep-copy. This increased heap usage and fragmentation is particularly problematic in small, resource-constrained devices such as WSN motes where there is little RAM.

To highlight the problem, consider a situation where data is being operated on by a pipeline of actors where each actor accepts data, does some processing, and forwards the data onwards, Figure 3.3. As the data is sent from one actor to another it will be copied, even though each actor will simply wait for the next piece of data and has no need to keep a reference to the processed data. A better situation would be where the sending actor would simply *hand-over* the data. Here no duplicate would be made, removing the increased heap usage and fragmentation. This is the purpose of movability.

This section describes the semantic model behind the movable memory space, its inclusion within Ensemble, and the compiletime analysis required to enforce the rules of movability.

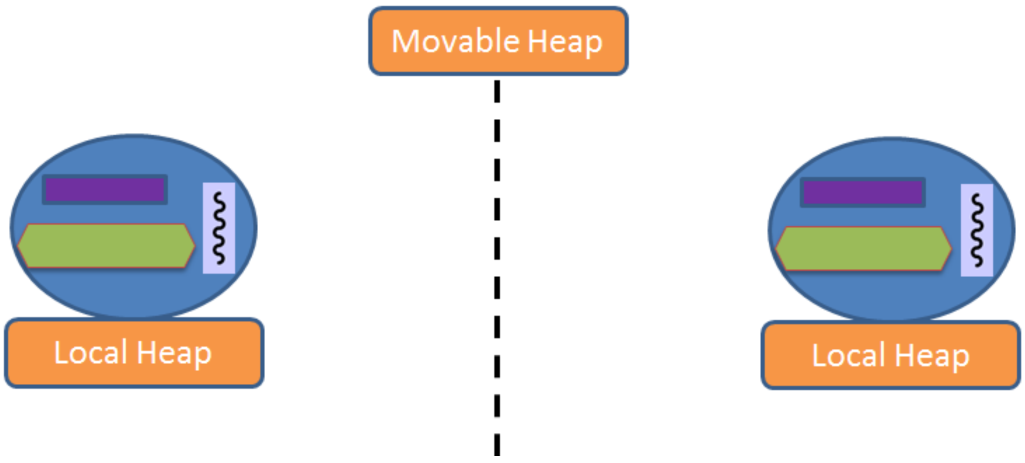


Figure 3.4: Actor Local and Movable Heaps

```
// data allocation from actor heap
alpha = new [1024]integer of 0;

// data allocation from movable heap
beta = mov new [1024]integer of 0;
```

Listing 3.6: Allocating Memory From Local and Movable Heap

3.2.1 Movability

To address increased heap usage and fragmentation, an additional memory space is proposed: the movable heap. In addition to the local heap associated with each actor, the purpose of the movable heap is that any memory allocated from it will be *unique* amongst all actors, such that data allocated from this heap which is sent along channels will not need to be duplicated. Figure 3.4 shows the **conceptual** model of the heaps in Ensemble.

Movability in Ensemble is an **optional** feature of the type system. It is not an annotation. It enables data to be allocated from the movable data heap, by adding the `mov` keyword in front of the `new` keyword during allocation, as shown in Listing 3.6. If used inappropriately, compiletime errors are provided to indicate where and why an error has occurred. The primary purpose is to reduce memory consumption and fragmentation.

The concept of unique data is a simplification of the memory spaces which are seen in the Rust programming language[2], which has a similar memory and concurrency model. Unlike Rust, Ensemble has a much less static programming model; Rust channel topologies may not be reconfigured at runtime. Also, Ensemble requires much less intervention from the programmer to indicate what is movable; Ensemble requires a single keyword, whereas Rust requires multiple keywords for the different memory spaces. As Ensemble actors are not

[2]rust-lang.org - Accessed Feburary 2015

only encapsulated, but also the core unit of abstraction in the language, there is truly no shared state at any point within an Ensemble application.

3.2.2 Ensemble Move Semantics

As well as defining the semantics of movable memory, it is important to describe the interaction of movable and non-movable memory. To preserve the movable semantics, some limitations must be placed on what is possible. In the following, “moved” refers to a piece of data which can be sent across a channel without being duplicated. Also, the following discussion describes in great detail the assumptions and use cases of movability. During actual development of an application, much of the burden of understanding any error cases is alleviated by the compiler, which describes how and where an error has been detected.

By using a single addition to the type systems, as well as describing at compiletime via errors which explain where and how an alias has been violated, control over moveability is offered to the user in a straight forward and tractable way. Ultimately, movability is an optimisation and is not required for the correct operation of an application, and can be removed without effecting the logic of an application.

Structures

The semantics for struct data types in Ensemble is to recursively apply the move semantics for references to heap memory. For example, a struct which has been marked as movable will have all primitive fields moved, and the references within the struct will have their movability checked to determine whether to deep-copy or move the reference. The following describes this process for the cases in Listing 3.7.

Case 1 - Allocation from Local Heap A duplicate is made of $simpleA$, and is sent on the channel. The sending and receiving actor will have a unique version of the data.

Case 2 - Allocation from Movable Heap The reference to $simpleB$ is sent. The sender must reassign a value to $simpleB$ before access, otherwise a compiler error is generated. The receiver has a unique copy of $simpleB$.

Case 3 - Allocation from Local Heap A duplicate is made of $complexA$, including the data pointed to by $simpleA$, and is sent on the channel. The sending and receiving actor will have a unique version of the data.

Case 4 - Allocation from Local and Movable Heap A duplicate is made of $complexB$ but not $simpleB$, as it is allocated from the movable heap. The duplicated $complexB$ with a reference to $simpleB$ is sent on the channel. $complexB$ may be accessed by the sender, but

```
type simpleStruct is struct(integer i);
type complexStruct is struct(integer i ; simpleStruct s);
...
actor Sensor presents ISensor {
  simpleA = new simpleStruct(1);
  simpleB = mov new simpleStruct(10);

  complexA = new complexStruct(1, simpleA);
  complexB = new complexStruct(10, simpleB);
  complexC = mov new complexStruct(10, simpleA);
  complexD = mov new complexStruct(10, simpleB);

  behaviour {
    // Case 1
    send simpleA on chan;

    // Case 2
    send simpleB on chan;

    // Case 3
    send complexA on chan;

    // Case 4
    send complexB on chan;

    //Case 5
    send complexC on chan;

    // Case 6
    send complexD on chan;
  }
}
```

Listing 3.7: Struct Movability Example

$simpleB$ must first be assigned to, otherwise a compiletime error is generated. The receiver has a unique copy of the $complexB$ and any internal references.

Case 5 - Allocation from Local and Movable Heap The reference to $complexC$ is sent, with a deep-copy of $simpleA$ as it was not allocated from the movable heap. The sender must assign a value to $complexC$ before access otherwise a compiletime error is generated. Not that $simpleA$ may have been aliased before being sent. Even though $complexC$ must be assigned to before being accessed in the sender, a duplicate must be made of $simpleA$ as there may be other references to it. Figure 3.5 illustrates the state of references to the local and movable heaps for this example.

Case 6 - Allocation from Movable Heap A reference to $complexD$ is sent. The sender must reassign a value to $complexD$ before access, otherwise a compiler error is generated. This reassignment will also remove the reference to the internal $simpleB$ in the sender. If there were other references to $simpleB$ in the sender which were accessed after transmission,

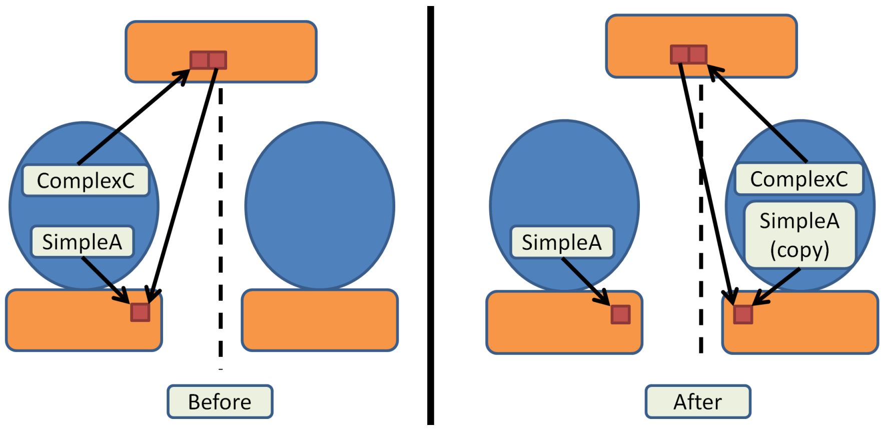


Figure 3.5: Actor and Movable Heaps Before and After $complexC$ is Sent Across a Channel

without being reassigned to, the compiler would generate an error. The receiver has a unique copy of $complexD$ and $simpleB$.

Arrays

A flexible array is one in which the size is unknown at compiletime - i.e. the array size can be initialised by an expression whose value isn't known until runtime. The presence of flexible arrays in Ensemble creates a problem for the movability analysis in determining how many references exist to an array element. Furthermore, array indices may be arbitrary expressions, making it infeasible to determine at compiletime which array element is being accessed; this is an issue when determining whether movable memory is being accessed after having been moved.

To handle the complexity, restrictions are placed on arrays, and extensions to the existing array creation mechanisms are provided. In the following discussion we assume an array element is a reference to a heap-allocated object. When declaring an array, a value or instance is normally passed, with each element being initialised to reference this value. In order to support movability, a template object may be provided to the new operator when declaring an array (lines 3 & 6, Listing 3.8). This creates a distinct object for each array element, rather than all elements referencing the same object.

For non-movable arrays, reference-counting of the array **elements** is performed, and the array is deep-copied on send operations. For movable arrays, the **array** is reference-counted and individual elements have the same count as the entire array. In other words, assigning from a moveable array element will increment the count of *the array*, not the single element. In this way, alisases between a variable and an array element are equivalent to the variable aliasing the entire array, for the purpose of ensuring the correct operation of movability at compiletime. Hence, sending a variable which aliases a movable array's element will invalidate the entire array. Equally, sending a dereferenced element of a movable array

```
type simpleStruct is struct(integer i);
...
a = new simpleStruct[n] of simpleStruct(0);
c = mov new simpleStruct(2);

b = mov new simpleStruct[n] of simpleStruct(0);
d = new simpleStruct(2);

// error, a is array of references to non-movable memory
a[n] := c;

// error, b is array of references to movable memory
b[n] := d;
```

Listing 3.8: Cannot assign movable memory to non-movable array or vice versa

(`send x[i] on output;`) will invalidate the entire array (`x`). When a movable array is sent, only the reference is passed and no deep copy is made.

By default, it is illegal to make reference assignments to references in movable arrays from non-movable references, and vice versa. This is necessary as it would not be possible to determine which are (non-)movable at compiletime. Hence, only arrays created in the movable heap (using `mov`) are considered to contain references to movable memory. Consequently, all elements of the array must be movable. Listing 3.8 shows that assignments to array elements of objects in opposing memory spaces results in a compiletime error. Note that the assignment, `c := a[n]`, would be valid as this will decrement the reference count on the movable object `c` referenced before the assignment, and increment the reference count on the non-movable object `a[n]` references. This is also true for `d := b[n]`.

It is possible to assign movable memory to non-movable memory or vice versa via the explicit use of the `copy` keyword. This keyword will safely convert movable data into non-movable or vice versa. Previously, `a[n] := c` would generate a compiletime error, as the two references point at different memory. Should it be appropriate, the developer may instead write `a[n] := copy c`. This will duplicate the data pointed to by `c`, and conceptually convert it to be of the same type of `a[n]`. This is true regardless of which side is movable or non-movable. This use of copy is a way to use duplication in a localised way.

Channels

In order to declare the memory of an object being received across a channel as movable, the definition of the `in` channel should contain the `mov` keyword, Listing 3.9 line 2. An actor who uses this interface will use the `receive` or `select` statements in the regular manner with the `mov_input` channel. The only difference is that the data received will be contained

```
type ImovableReceive is interface(
  in mov simpleStruct mov_input; // <-- movable heap object
  out simpleStruct output;
  out integer int_output
)
```

Listing 3.9: Declaration of an `in` channel with movable data

```
a = mov new Foo;
behaviour {
  send a on chan;
  stop;
}
```

Listing 3.10: Ill-formed Behaviour Clause

within the movable memory space, and the compiletime analysis will track the data to ensure that the object is only ever moved once. It is important to note that it does not matter whether the data from the sender was in the movable heap or not. The data was either in the local heap and deep copied, or the data was in the movable heap and is unique. In either case, the data is now entirely owned by the receiver, and state encapsulation is preserved.

It is also important to note that although the data which a channel conveys may be marked as movable, the channel declared in the interface itself cannot be marked as movable. As the channels of the interface are associated with an actor when it is defined, they are permanent and cannot be removed. To do so would complicate the logic of discovering actors described in Section 3.4.3 due to the changing type of the actor. This said, any channels declared at runtime can be allocated from the movable heap, and be sent without deep-copy. This is an optimisation applied to the accelerator-based applications discussed in Section 5.2.

Effect on Behaviour Clause

Movable memory references which are sent within the behaviour clause, either directly, or indirectly via a procedure call, must be initialised or re-initialised at some point within the behaviour clause without any enclosing control flow construct. The use of the `stop` statement within the behaviour clause does not negate this requirement. For example, Listing 3.10 shows an ill-formed behaviour clause. Even though `a` will never be sent twice because of the presence of the `stop` statement, the program is not valid since, in general, the presence of a `stop` statement does not guarantee behaviour termination after one iteration (see Listing 3.11).

```
a = mov new Foo;
behaviour {
  send a on chan;
  receive v from intchan;
  if v == 1 then {
    stop;
  }
}
```

Listing 3.11: Stop does not guarantee correct execution

3.2.3 Approach

In order to enforce the semantics of movability, an application is represented as a series of Control Flow Graphs (CFG) [103]. This is done as it is a convenient representation upon which to perform the analysis. A CFG represents a procedure as a directed graph, $G = (N, E)$, where the nodes, N, contain the procedure's instructions, and the edges, E, represent control flow through the application. Here, control flow refers to the constructs of the source language that determine whether or not a section of code is executed. Traditional control flow constructs are the `if`, `while`, and `for` statements. The CFG, by convention, has a unique entry node, and a unique exit node.

The nodes in a CFG represent a *basic block*, which is a sequence of instructions where the only branching code occurs at the end of the block. In other words, if a program enters a block during execution, then all of the instructions inside the block are executed. Each block has zero or more successor blocks, and zero or more predecessor blocks represented by the directed edges of the CFG.

A Call Graph (CG) [104], or call multigraph [105], is a directed graph, $G = (N, E)$, where the nodes of the graph, N, represent procedures in the application, and the edges of the graph, E, represent calls from the source node to the destination node. It is a useful representation of the relationship between caller and callee, and serves as an information repository on procedures during analyses.

The *supergraph* [106] representation of a program connects the CFGs of callers and callees, using *interprocedural edges*.

The following section gives an overview of the steps required to ensure the correct and safe usage of data allocated from the movable heap at compiletime. A formal description of the movability analysis can be found in Appendix A.

3.2.4 Analysis

Abstractly, the core of the analysis is concerned with keeping track of which movable variables have been sent across channels. Should such a variable or one of its aliases be sent across a channel and then subsequently accessed before it has been reassigned to, an error is generated.

This is achieved by performing data flow analysis in the front end of the compiler, after the type checking phase. The analysis is done in two passes. The first pass is the *alias* analysis which computes for all application variables their set of potential but unknown aliases (may-aliases) at each program point. This information is then used in the second pass which performs movability tracking and error detection on the program to determine whether or not any movable variables have been accessed inappropriately.

Before the analysis is performed, the Ensemble application is represented using the IR described in Section 3.2.5. This was done to have a convenient representation of an application for this analysis, rather than overloading the existing Abstract Syntax Tree (AST). Using this representation, both analyses perform a reverse pre-order traversal of the graph representing the Ensemble application. This is done as it is a simple way to remove back edges, and represents a linearisation of control flows.

Alias Analysis

The alias analysis is used to track all aliases which could potentially refer to movable memory. During the analysis, each point in the application has an associated In and Out set which is used to hold the variables which are present on input to and out from each program point respectively. For the intraprocedural analysis, the In and Out sets are modified by assignment to references and call sites. For the interprocedural analysis, the call and return nodes generate and remove aliases between actual and formal reference parameters, respectively.

The interprocedural analysis uses the call-string approach [107]. The key concept in the call-strings approach is to maintain a *token* stack, where each token represents a procedure call which has not yet returned. The tokens on the stack are known as the *call string*, or the *calling context*.

If a call node is encountered during the analysis, it is appended to the context and the data flow information is propagated along the call edge to the entry node of the CFG representing the callee. When a return node is encountered, the corresponding call node is removed from the context, and the data flow information is propagated along the return edge to return node of the CFG representing the caller. For all other nodes in the supergraph the alias sets are updated by the intraprocedural data, and the context (stack) is unchanged.

For programming languages without recursion the maximum length of the calling context is bounded by the depth of the call graph. Therefore, if the original data flow problem terminates, so too does the interprocedural extension of the problem. As Ensemble supports a limited form of recursion, where a procedure may only recurse on itself, it is sufficient to evaluate one level of recursion and return when the procedure attempts to recurse again. This is because each subsequent level of recursion (and analysis) would duplicate any alias modifications present in the first two levels. This would not be possible if one procedure were able to call another recursively.

At the end of the first pass, a set of aliases for all application points has been built.

Move Analysis

The move analysis is concerned with keeping track of which variables have been sent across channels. Should a variable or one of its aliases be sent across a channel and then subsequently accessed before it has been reassigned to, an error is generated. This access could also include attempting to send the data across a channel.

The analysis itself consists of performing the reverse pre-order traversal of the graph and applying these rules for each node.

The combination of these analyses enable the safe usage of movable types at runtime via compiletime guarantees.

3.2.5 Intermediate Representation

In order to perform the analysis, the AST which is generated by the compiler frontend is transformed into an IR which provides the necessary information for the analysis, without complicating the implementation of the AST.

Programs, Actors, and Functions

The IR defines a representation for the entire application, enabling the analysis to obtain information about a variety of application properties. The top level representation describes an entire Ensemble application; it holds mappings for all actors, globally defined procedures, and interfaces defined within the program. Additionally, it maintains the program CG and the CFG for a stage's boot clause (actor initialisation, connection statements, etc).

Kind	Description	Form	Uses
assign	Assignment	*a := b*	uses(a), b, uses(b)
call	Procedure call	$f(a_1, a_2, \ldots)$	a_1, uses(a_1), a_2, uses(a_2), ...
cjump	Conditional jump	*cjump cond,B1,B2*	$cond$, uses($cond$)
connect	Connect statement	*connect a to b*	a, uses(a), b, uses(b)
decl	Declaration statement	$a = b$	b, uses(b)
disconnect	Disconnect statement	*disconnect a*	a, uses(a)
jump	Unconditional jump	*jump B1*	-
nop	No operation	*nop*	-
recv	Receive statement	*receive a from b*	b, uses(b)
ret	Return statement	*return r*	r, uses(r)
send	Send statement	*send a on b*	a, uses(a), b, uses(b)
stop	Stop statement	*stop*	-

Table 3.2: List of available IR instructions and their internal uses.

Instructions

To simplify the representation for analysis, the IR was defined as closely modelling Ensemble constructs, but deconstructing control flow constructs into simple conditional and unconditional jump instructions. Expressions are decomposed so that at most three operands appear in one instruction along with a binary operator. Some expressions, such as array and structure dereference expressions, are not deconstructed to simplify tracking composite movables. Having access to the left and right hand side of an expression simplifies the comparison of movable status. This is necessary to restrict array assignments to comparable memory types (see Section 3.2.2), and also because the type checker cannot determine this for all application situations, such as the movable status of a value returned by a function call.

The IR instructions are collected together into basic blocks. The exit point at the end of the block is either a conditional or unconditional jump to at most two blocks. Each block may have any number of predecessor blocks. Table 3.2 gives an overview of the instructions in the intermediate representation. Figure 3.6a shows an Ensemble code snippet, Figure 3.6b shows the corresponding Ensemble IR upon transformation (basic block labels have been added for clarity), and Figure 3.6c displays the corresponding control flow graph segment.

Value Representation

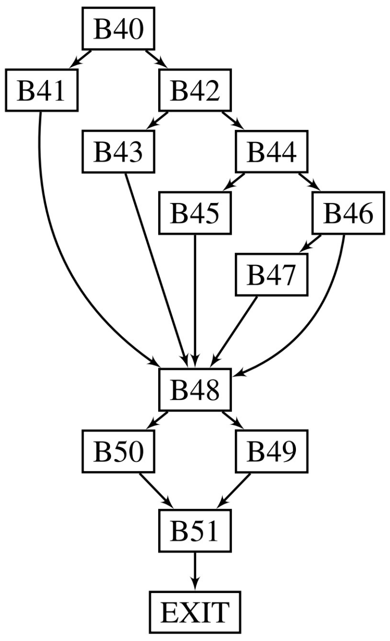
The IR also represents application variables and values. As the data flow analyses require a representation for application variables, it was convenient to use the same representation within IR instructions as for the data flow values rather than the more commonly used bit-vector representation. The additional space overhead within the data flow sets is not onerous,

```
if val == 0 then {
        printString("Rec:␣got␣tic\n");
} else if val == 1 then {
        printString("Rec:␣got␣toc\n");
} else if val == 2 then {
        printString("Rec:␣got␣tac\n");
} else if val == 3 then {
        printString("Rec:␣got␣def\n");
}
if val == 0 then {
        printString("Rec:␣got␣tic\n");
} else {
        printString("Rec:␣got␣toc\n");
}
```

(a) Ensemble Source

```
B40:
t20 := val == 0
if t20 then goto B41 else goto B42
B41:
printString("Rec:␣got␣tic\n")
goto B48
B42:
t21 := val == 1
if t21 then goto B43 else goto B44
B43:
printString("Rec:␣got␣toc\n")
goto B48
B44:
t22 := val == 2
if t22 then goto B45 else goto B46
B45:
printString("Rec:␣got␣tac\n")
goto B48
B46:
t23 := val == 3
if t23 then goto B47 else goto B48
B47:
printString("Rec:␣got␣def\n")
goto B48
B48:
t24 := val == 0
if t24 then goto B49 else goto B50
B49:
printString("Rec:␣got␣tic\n")
goto B51
B50:
printString("Rec:␣got␣toc\n")
goto B51
B51:
goto EXIT
```

(b) Ensemble IR (Textual)



(c) Ensemble IR (Graphical)

Figure 3.6: Transformation from Ensemble to Ensemble IR

Description	Form	Uses
Array construction	*new <type>* $[n_1] \cdots [n_k]$ *of* $init$	uses(n_1), ..., uses(n_k), uses($init$)
Binary operation	*a op b*	a, b, uses(a), uses(b)
Channel construction	*new out integer*	-
Structure construction	*new <type>* $(v_1, v_2, \ldots)$	uses(v_1), uses(v_2), ...
Copy Operation	copy v	v, uses(v)
Array subscript expression	$v_1[v_2]$	abs_base(v_1), uses(v_1), v_2, uses(v_2)
Structure field expression	a.x	abs_base(a), uses(a)
Constant integer	4	-
Identifier	v	-
Literal	v	-
Channel selection (select statement)	*select* $\{ticker, tocker, tacker\}$	$ticker$, $ticker$, $tacker$
IR Temporary	t4	-
Unary operation	*unary_op* v	v, uses(v)
Function call	$f(a_1, a_2, \ldots)$	a_1, uses(a_1), a_2, uses(a_2), ...

Table 3.3: List of available values and their internal uses.

and provides efficient access to various properties of application variables that can be used in the analysis to generate detailed error messages and obtain memory type information.

Table 3.3 summarises the possible values that can appear in an Ensemble application. Here, the *abs_base()* is used to find the absolute base value in array subscript and struct field expressions. Notice that a call is represented both as an IR instruction and a value. The value encodes Ensemble procedures that return values, whereas an IR call instruction corresponds to a void procedure. To distinguish between the two, we use the term *function* to describe a routine returning a result and *procedure* to describe a routine returning void.

3.3 Integration of Actors and Accelerator-Based Concurrency

Due to power consumption, heat dissipation, and clock propagation limits, modern hardware architectures are now designed with many, concurrent processing elements, as opposed to single processing elements with increasing clock rates; examples of such architectures include GPUs and multicore CPUs. These hardware platforms are designed to provide the user with multiple physical threads of execution, thus enabling many computations to occur simultaneously.

On single CPU architectures, threads have traditionally been used to enable parallel execution. However, due to the different nature of GPU hardware architectures, a number of different programming techniques are used. OpenCL is a standardised programming framework available for the main GPU vendors (NVIDIA and AMD), as well as other parallel hardware architectures.

While OpenCL enables access to these architectures, there are three main limitations. Firstly, the user is required to write large amounts of boilerplate code to create the OpenCL environment for a particular calculation. Secondly, the programming style requires explicit data movement between the host CPU and the OpenCL device; this requires flattening multi-dimensional arrays and structures of non-primitive types. Thirdly, the language and style used to program the device is often different from the programming language being used on the host. A similar argument can be made against the CUDA framework; since CUDA is only available on NVIDIA hardware, this work explores the more broadly applicable OpenCL.

This section describes the first application of actor-based programming to accelerator-based concurrency at the language level by including the OpenCL framework within the Ensemble programming language. Natively, Ensemble applications can run concurrently in a task-parallel context, where actors embody the units of computation. Given the overall hypothesis of this dissertation, the combination of actors and accelerator-based programming shows that moving from low-level C code to a concurrent, shared-nothing, high-level actor programming model simplifies the use of OpenCL by providing appropriate structuring, thus enabling greater access to high-performance and heterogeneous computing.

3.3.1 OpenCL

OpenCL is a programming framework for heterogeneous and parallel computing. It is standardised, and is managed by the Khronos working group[3]. In OpenCL, users are required to think in terms of host and device code, where a host is a coordinator application on the CPU, and a device is an *accelerator*. An accelerator may be a CPU, GPU, Field-Programmable Gate Array (FPGA), or co-processor such as the Xeon Phi [79].

OpenCL Configuration

In OpenCL, the host is tasked with setting up, dispatching, and collecting results from a device. OpenCL is accessed through an API, which enables relatively low-level access to data types and functions in order to program and interact with one or more accelerators.

Creating an OpenCL environment consists of first querying the hardware at runtime to determine the available vendor *platforms*, and the *devices* available in each platform. Platforms are essentially drivers provided by the hardware vendor, and the devices represent the actual accelerators. Then, a `context` must be created. A context is an umbrella structure that holds the device(s) to be used, as well as other runtime software constructs. A `command_queue` is then associated with each device and placed within the context.

[3]https://www.khronos.org/opencl/ - Accessed October 2014

A command_queue is used to issue commands to a device. Commands include device queries, memory management operations, and kernel (Section 3.3.1) invocations. After this, a user creates a program with the kernel source file, and compiles it at runtime. The specific function to be executed within the compiled source is then used to create the kernel object. At this point the OpenCL environment has been constructed and is ready to be configured and used for a specific computation.

From here the user allocates memory on the device and then copies host data into this memory. The device memory is then associated with the correct position in the kernel arguments. Then, the number of dimensions upon which the kernel should work is calculated, and the kernel is launched on the device, with this information, via the command_queue. Usually, the host then blocks attempting to read data back from the device once it has finished its computation. Once all computation is complete and the device is no longer required, there are appropriate destructor functions.

The device itself is treated simply as a functional unit. Data and code are passed to the device, the device executes this code, and the results are read back by the host.

Kernels

A device runs a special piece of code known as a *kernel*. An OpenCL kernel is written in a C-like syntax and represents the logic of a single thread. The number and groupings of threads are supplied during the configuration stage on the host. These values are known as the local and global worksizes, and are used to optimise the allocation of threads to the underlying hardware for a given dataset.

Within a kernel, the currently executing thread may be identified via the API. This can be used to customise application logic. The kernel is expressed as a function with parameters. Information for the actual computation is passed to this function as arguments by the host.

The OpenCL model uses a memory hierarchy in which memory is split into global, local, private, and constant regions. This is a direct mapping to the hardware configuration of memory found in GPUs, however the same model is applied to all hardware devices. Global memory is shared amongst all threads, local memory is shared between a specified group of threads, and private memory is specific to a thread. Global and local memory are subject to unsynchronised modifications, although there are mechanisms to synchronise access. Constant memory is shared by all threads, but is read only. A simple OpenCL kernel is shown in Listing 3.12.

The similarity between the isolated nature of computation in actor-based programming and the OpenCL model is strong. Both require explicit data movement between loci of computation. In Ensemble, the channel mechanism provides a natural way to facilitate this without

```
__kernel void square(__global float* input,
                     __global float* output,
                     const unsigned int count){
    int i = get_global_id(0);
    if(i < count)
        output[i] = input[i] * input[i];
}
```

Listing 3.12: OpenCL Kernel to Compute the Square of An Input Array

the need to go *outside the language* or obfuscate existing code, consequently, OpenCL kernels are represented as actors. The requirement for separate programming of host and device also matches the distinction between actors. A full example of matrix multiplication in Ensemble is shown in Listing 3.13 and is referred to throughout this section. Here, the green colour represents additions to the language, and light yellow shows compiler-enforced structure. These highlighted sections represent the only modifications to the language.

3.3.2 Language Model

In Ensemble, an actor is marked as being a kernel by adding the `opencl` keyword to its definition (line 21). This tells the compiler that the actor's interface should only contain a single channel, and that a slightly different structure is expected within the behaviour clause. After this keyword, an optional set of angle brackets may be supplied by the user to specify certain configuration information to the runtime. This information includes one of the `GPU`, `CPU`, or `ACCELERATOR` language keywords in order to specify the `device_type` to be used by this actor: a GPU, CPU, or generic accelerator type. This last case is used for devices such as the Xeon Phi [79]. Also, the `device_index` is used to specify the device to be used. This is often required to distinguish between multiple devices within the same category. Both of these configuration parameters are optional, may be used independently of each other, and may be specified in any order. Should the `device_type` not be specified, the default device dictated by the OpenCL runtime is used. Should the `device_index` not be specified, the 0^{th} device is chosen. In this way, kernels are integrated into the language, as opposed to a separate source file in a different language.

The single channel required in the actors interface conveys an `opencl struct` type as defined by the developer (lines 6-11). This is a normal `struct` except that the `opencl` keyword tells the compiler that the fields of the `struct` should contain two integer arrays, an `in` channel, and an `out` channel. The arrays convey the local and global worksizes required to dispatch the OpenCL kernel, as discussed in Section 3.3.1. The two channels are the input and output channels for the data to and from the kernel, respectively. The channels

```
type data_t is struct (
    real [][] a;
    real [][] b;
    real [][] result
)
type settings_t is opencl struct (
    integer [] worksize;
    integer [] groupsize;
    in data_t input;
    out real [][] output
)
type dispatchI is interface(
  out settings_t requests;
  out data_t dout;
  in real[][] din
)
type mulI is interface(
  in settings_t requests
)
stage home{
  opencl <device_index=0, device_type=CPU> actor Multiply presents mulI {
      constructor() {}
      behaviour {
          receive req from requests;
          receive d from req.input;
          x = get_global_id(0);
          y = get_global_id(1);
          dim = get_global_size(0);
          c = 0.0;
          for i = 0 .. (dim-1) do {
              c := c + (d.a[y][i]) * (d.b[i][x]);
          }
          d.result[x][y] := c;
          send d.result on req.output;
      }
  }
  actor Dispatch presents dispatchI{
      constructor() {}
      behaviour {
          s = 1024;
          ws = new integer[2] of s;
          gs = new integer[2] of 0;
          i = new in data_t;
          o = new out real[][];
          connect dout to i;
          connect o to din;

          ocl_struct = new settings_t(ws,gs,i,o);
          d = generate_data(s);

          send ocl_struct on requests;
          send d on dout;
          receive result from din;
      }
  }
  boot{
    d = new Dispatch();
    m = new Multiply();
    connect d.requests to m.requests;
  }
}
```

Listing 3.13: GPU Matrix Multiplication in Ensemble

are created and configured in the host actor (lines 43-46), sent to the kernel actor via the input channel (line 51), and then the data is sent (line 52). The host then waits for data from the actor (line 53). This is an example of the dynamic nature of Ensemble channels, and enables any type of data to be conveyed to and from the kernel actor safely without requiring any extra compiler analysis beyond normal language processing.

Within the behaviour loop of the kernel actor, it is required that the first two statements are receive statements (lines 24-25). This first statement receives an `opencl struct` instance. This enables the runtime to prepare the kernel for dispatch to the device with the appropriate dimensions by using the `worksize` and `groupsize` values. The second statement is used to receive the data that the kernel will process. It is also required that the last statement is a send statement (line 34). This is used to send the processed data onwards. The send statement is also used as a marker, with all statements between the second receive and send statements representing the OpenCL kernel (lines 26-33). The standard set of OpenCL calls natively available in a kernel are also available within an OpenCL behaviour clause between these markers (lines 26, 27, 28), including the math functions. Also, the `global`, `local`, and `private` memory modifiers are available, enabling the developer to take full advantage of the different memory regions which are available in the OpenCL model to improve performance.

One of the key advantages of embedding OpenCL within the language is the ability to preserve multi-dimensional array and structure dereferencing. Currently, OpenCL requires that arrays and structures containing pointers are flattened when being passed to a kernel. In Ensemble, this process is automated, leaving the users with normal dereferencing of such data types within the kernel. This also has the advantage of providing the user with warnings and errors for kernels at compiletime, rather than having to wait until runtime kernel compilation. Furthermore, as all the OpenCL actors are connected by channels, should the user wish to change the device upon which the OpenCL actor should run, the language only requires that the device type be modified in the actor definition. No other change is required. Also, should the developer wish to use a different kernel or a different device at runtime, all that is required is to reconnect the configuration channel to an appropriate kernel actor's configuration channel. Should the local stage not support OpenCL, kernel actors may be `spawned` at remote stages, enabling more possible (re)configurations. This is discussed in Section 3.4.4.

By tagging an actor as an OpenCL kernel, a user must still write parallel code within the behaviour clause, however by integrating this into the language model, the process is substantially simplified compared to a C version, see Section 5.2.3.

Movability

One of the greatest costs in performance for OpenCL applications is data movement. Specifically, the movement of data between the host and the device during the execution of an application. This cost is often mitigated by leaving data on an accelerator for as long as possible. This concept is in direct conflict with the shared-nothing semantics of the actor programming model.

The presence of the movable heap in Ensemble offers a solution to this problem. In addition to removing the need for data duplication when sending data between actors, moveable data can be used to mark GPU data for lazy evaluation. Hence, if movable data is sent from one kernel-actor to another on the same device, without being accessed in the interim, the data will remain on the device. This is discussed fully in Section 4.5.2. Hence, movability enables this common GPU optimisation in the actor programming model without violating the shared-nothing semantics.

3.3.3 Compiler Modifications

The compiler was modified to apply slightly different rules to an OpenCL actor, enforcing the structure described in Section 3.3.2. The channel operations mirror the explicit data movement required by OpenCL, and it is the channel operations that dictate the creation of device buffers and the movement of data between the host and the device. `struct` values are flattened so that each field is sent separately, with the compiler generating the appropriate code within the kernel to manage this. Multi-dimensional arrays are flattened to single dimensional arrays. Again, the compiler will generate appropriate kernel code to manage this. Primitive values are sent as 1D-arrays of one element so as to ensure updates within the kernel are applied to the value. Passing a pointer to the host variable is not an option. A potential optimisation here is to wrap all passed primitive variables in a single array.

3.3.4 Execution Model

Figure 3.7 shows the execution model when using OpenCl with Ensemble. Rather than the actor running on the device directly, an OpenCL actor is compiled into a bytecode representation of the actor in the normal way. The Ensemble compilation process and runtime representation is discussed in Section 4.2. A C representation of the code identified as the kernel is generated, and stored as a string within the actor's bytecode. Should the kernel actor contain functions defined in other code sections, the compiler will generate C equivalents within this string. This is completely hidden from the developer. The bytecode is

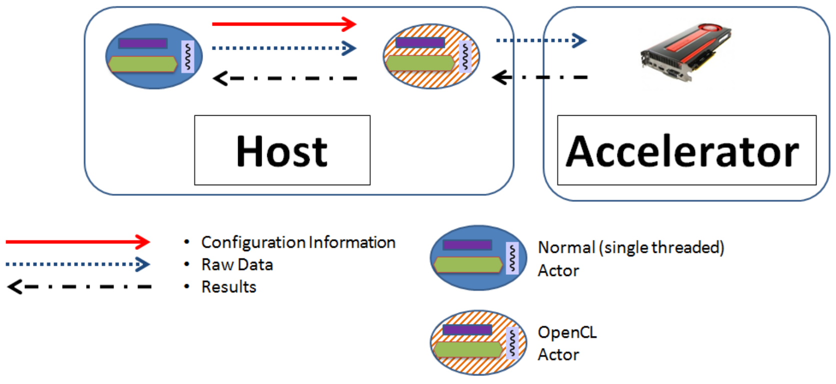


Figure 3.7: OpenCL in Ensemble Execution Model

interpreted as normal, and acts as the host in the traditional OpenCL sense. This actor handles the incoming and outgoing channel communications, as well as preparing, launching, and collecting data from the kernel. This also enables multiple kernels to execute on a single device.

3.4 Adaptability in Ensemble

Given the encapsulation of actors, and the use of message passing to facilitate communication, the actor model is perfectly suited to transparently enable distributed programming. Although this is true of other languages, Ensemble expresses each phase of the adaptation process, from discovery to reconfiguration, entirely within the language, without the need for external configuration files or out-of-language APIs (Section 2.2). Also, whereas other actor-based languages will only support the creation of a new actor remotely, Ensemble also supports the strong migration of running actors from one stage to another across heterogeneous hardware platforms.

Adaptability, or runtime reconfiguration, in Ensemble applies to stages, actors, and channels. Adapting these entities consists of four distinct steps: defining unique attributes of language entities, defining queries to distinguish between these entities based on attributes, discovering these entities at runtime, and then using the discovered entities to adapt the runtime environment by creation, destruction, or reconfiguration. This section describes how adaptability is expressed entirely within the language; Section 4.4 provides a discussion of the runtime implementation of adaptability. Section 2.2, which contains a discussion of a number of different on-demand service discovery approaches, provides a background for the discussion in this section.

```
type property is struct(string key ; any value)
```

Listing 3.14: The Ensemble Property Type

```
// a query definition
query example_query(integer alpha; bool beta; string gamma){
  alpha > 67 and
  alpha < $remote_key_ex1 or
  (beta and $remote_key_ex2 != gamma) and
  can_run(example_actor);

  // a query instance
  example_instantiation = example_query(1, true, "test");
}
```

Listing 3.15: A Query Clause

3.4.1 Defining Unique Attributes

An actor is defined with a name and a set of interfaces. This name is used when creating new instances of actors as the compiler can guarantee that no two actors will have the same name. However, in a distributed context, where applications are compiled independently of each other, it is possible that there may be multiple actors defined with the same name. This is equally true of stages. It is important to note that Ensemble has no limitation on actors from different applications interacting with each other; indeed this is a likelihood given the runtime reconfiguration of actors. As similarly named actors may be operating within the same stage, names alone are not suitable to uniquely identify actors. To achieve this goal, the `property` type is used.

The `property` type represents a key-value pair consisting of a `string` for the key and an `any` for the value, Listing 3.14. This struct must be defined by the language as the compiler ensures that the `any` type holds a primitive value, otherwise an error is generated. The primitive value is required to simplify the formulation of queries, Section 3.4.2.

Having actors from different applications working within the same environment could be a possible security issue. Security considerations were mostly beyond the scope of this project, but there is a brief discussion in Section 4.7.

3.4.2 Queries

The first step in locating either actors or stages is to define a *query*. Listing 3.15 shows an example query definition and is referred to in this section. A query definition is similar to a procedure definition in that it has a name and an optional set of parameters, however, the body of a query definition only consists of a boolean expression.

Application of the boolean expression to the universe of actors (or stages) yields a set of actors (or stages); the set can be empty, consist of a single actor (or stage), or 2 or more actors (stages). The expression itself may only use primitive types, which includes the `string` and `stage` types; user-defined types may not be used as this would require the user to define comparison operators for each defined type. This decision was made to simplify the queries, and errors are reported at compiletime. The query parameters may also be used in the expression, so long as they are primitives. Parameters may only be compared against similarly typed variables or values, otherwise a compiletime error is generated. There are two exceptions to this rule: *remote keys* and the `can_run()` operator.

Remote keys are names which refer to the key field of properties which are expected to be found at the remote stage or actor, (lines 4 & 5). Remote keys are identified by names prefixed with the $ symbol. So as to remove any coupling between actors or stages, it is possible to use remote keys in a query which do not match properties defined in any stage or actor within the scope of the query definition. Hence, no compiletime error will be generated for comparisons against remote keys as the evaluation happens at runtime. Should the remote key not exist during evaluation, the relevant clause in the expression will return false, preserving the logic of the complete expression in a meaningful way. This has the advantage of not requiring any central design or coordination of queries or properties in order to guarantee legal operation, although it is the responsibility of the user to guarantee logically correct operation.

The `can_run()` operator is used to give the developer some advanced indication as to whether or not a stage will support an actor. The operator takes the name of a defined actor and returns `true` if the stage evaluating the query can execute the specified actor, otherwise `false`, (line 6).

A query definition is similar to a procedure definition, however, a query definition always *implicitly* returns a `query` type. A `query` type is the representation of an instance of a query definition. It is a first class entity in the language, which may be sent along channels or used in discovery.

Although there are languages, such as SQL, which are designed to specify queries, it was decided to use a language-specific format to specify the query. This was done to reduce the amount of extra effort required on the part of the developer, who would be required to learn SQL or some other domain specific language. Furthermore, it is not yet apparent that the Ensemble-based approach either lacks sufficient expressiveness or is overly verbose for the applications discussed in Chapter 5.

```
actor test presents Itest{
  props = new properties[1] of property("dummy", any(0));

  constructor(){
    publish props;
  }
  behaviour{
    // do something

    unpublish;

    // do something else

    receive other_props from input;

    publish other_props;
  }
}
```

Listing 3.16: Property Definition in Pseudocode

3.4.3 Discovery

Discovery is the process of locating either actors or stages based on the properties that they possess, which are evaluated by queries. This is different to approaches such as Scala, where actors are located by an IP address, or Salsa, where actors are located by name or IP address. Within Ensemble, discovery happens at runtime, and is completely controlled from within the language. Having all adaptation operations integrated within the language simplifies its use, expression, and comprehension. There is no need to supply lists of IP addresses, or hostnames in order to locate or reconfigure the language entities (Section 2.2). That said, it is required to pre-deploy stages, as they are not configurable from within the language. This is similar to the MPI approach, and is discussed in Section 4.3.2.

Actor Visibility

Before an actor is discovered it must explicitly have a set of properties associated with it. This is done by first creating and initialising an array of properties, and then linking this array with the enclosing actor via the `publish` statement, Listing 3.16. An actor must explicitly `publish` itself in order to be discovered, as actors are not visible by default. Note that an array of properties may be sent along a channel, rather than being constructed locally. Again, the `publish` statement is part of the language, and not part of a library. Once an actor publishes itself, it may be discovered by other actors.

The `unpublish` statement is used to make the calling actor hidden, and removes the associated properties. Should an actor wish to update its properties, it need only reissue a publish statement with the specified properties, rather than first unpublishing. It is important to note that it is not possible for one actor to publish or unpublish another, as to do so would violate the encapsulation of actors. This approach gives the user programmable control over how and in what way an actor is located at runtime.

An actor may only publish a set of unique properties. As property arrays are constructed at runtime, it is not possible to check this assertion at compiletime. Instead, the runtime will perform this check and generate an exception if the list does not consist of uniquely named properties. This is done to ensure correct runtime behaviour.

Actors were chosen as discoverable entities to enable the reconfiguration of channel connections. Equally, channels could have been chosen instead of actors, however, this would have increased the complexity of Ensemble applications, as each channel would require a set of associated properties. Also, accessing channels in this way provides a grouping mechanism which would not be present when accessing channels directly.

Stage Visibility

Unlike actors, `publish` and `unpublish` statements do not apply to stages as they are always visible. Instead of user-defined arrays of properties, a pre-defined set of properties are associated with each stage:

- **#NAME**: A string representing the name of this stage as defined in its definition.
- **#CORES**: An integer representing the number of CPU cores available at this stage.
- **#AVAIL_RAM**: An integer representing the current number of bytes of free RAM available at this stage.
- **#ACTORS**: An integer value representing the current number of actors resident on this stage.
- **#OPENCL**: A boolean value indicating if OpenCL actors are supported on this stage.
- **#DISTANCE**: An integer representing the physical distance between the querying and queried stage.
- **#CPU_LOAD**: An integer representing the current load across all CPUs as a percentage.
- **#HOSTNAME**: A string representing the name of the underlying physical platform.

- **#NET_ADDR**: A string representing the network address of the underlying physical platform.

These properties reflect attributes of the stage instance, as well as runtime values. For example, when queried, the `#AVAIL_RAM` property value will represent the amount of free RAM currently available in the system, rather than a pre-defined value. The `#DISTANCE` property is discussed further in Section 4.4.1. This was done to enable decisions to be made when considering load-balancing at runtime. The control of fine-grained load-balancing from within the language is beyond the current scope of this work, but is an area of future work. The current properties are exemplars of appropriate key-value pairs for a stage, and serve as a starting point for future work; they are not necessarily complete or correct. The decision to use static properties is discussed further in Section 3.4.6

Finding Actors

Once a query has been defined, published actors must be explicitly discovered via the language-defined `findActors()` procedure, Listing 3.17. This procedure takes an `interface` type and a `query` type. The decision to use a language-defined procedure was made as a simple quasi-English statement, as has been used for other operations in the language, was not found.

The searching actor is required to specify an interface to ensure that it will be able to safely and correctly access the channels of any discovered actors. Also, the specified interface acts as a filter, preventing actors who do not `present` such an interface from being eligible for discovery. In order to decouple the design and compilation of Ensemble applications, an actor is defined as supporting the specified interface if it *structurally* matches as opposed to name-based equivalence. The two approaches are discussed in Section 2.2.4. Here, structural matching was chosen as Ensemble applications are expected to be compiled independently and still interoperate, hence the functional correctness and type safety of structural equivalence was chosen. The implementation is discussed in Section 4.4.1. Although this approach does not guarantee semantic correctness, the `query` can be used to help provide a stronger guarantee than structural matching alone.

The expression which the specified `query` represents is evaluated against the properties of actors which are visible in both the local and remote stages; it is possible to identify only local actors using the desired (local) stage's name within the query definition. Actors where the entire expression within the `query` evaluates to true are considered eligible, and are returned to the searching actor. The combination of the interface, properties, and queries can enable a desired actor or set of actors to be accurately pinpointed.

```
proc findActors(interface i; query q) : typeof<i>[];
```

Listing 3.17: **Pseudocode** for the findStages Prototype

```
// locating a stage via discovery
proc findStages(query q) : stage[];

// creating a stage reference manually
a_stage = new stage("192.168.0.1");
b_stage = new stage("host.gla.ac.uk);
```

Listing 3.18: findStages Prototype

Once the search is complete, an array of `interfaces` of the type specified is returned by the `findActors()` procedure to the calling actor. Each interface is a reference to a unique actor. The main difference between an actor reference created by `new` and one of these references being that a reference to a newly created actor can access all channels that the actor presents, whereas a reference to a discovered actor only enables access to the channels specified in the interface used to find it. If no actors are found, this array will be empty.

Ensemble does not support generics, and so creating an array of this type is a special case handled by the compiler. These interfaces have the type of the interface passed to the `findActors()` procedure. Also, as the type and number of available channels is known at compiletime, the compiler can generate errors if the actor reference itself or the dereferenced channels are used improperly. These interfaces may be used in the same way as any other interface (Section 3.1.3). It is important to note that located actors may have more channels than specified in the interface used for discovery. Although actors may `present` multiple interfaces, they are only considered eligible should the specified interface match at least one of the presented interfaces.

Currently, there is no way of distinguishing between the actors which have been returned from a discovery. The only certainty is that all references support the specified interface, and meet the conditions defined in the query. In situations where a more specific choice needs to be made, a search can be repeated using a query with more precise requirements.

Another approach would be to provide a function to the discovery process which would apply an ordering to the results. The sorting condition would refer to the properties referenced by the boolean expression in the query. This is the approach taken in the DPL language [108], and is a potential avenue of future work.

Finding Stages

The process of locating stages is similar to and simpler than locating actors. In order to discover a stage the `findStages()` procedure is used, Listing 3.18. Like actors, this procedure takes a `query` type, but does not require an interface as stages do not use interfaces or channels. This procedure will return an array representing the stages which were found and that satisfied the specified query. If no matching stages were found, this array will be empty. Also like actors, there is not currently any way of distinguishing between the stages which have been returned. The only certainty is that all stages in the array meet the conditions specified in the query. Again, to find a specific stage, a more precise query could be reissued, or a sorting condition could be supplied.

As well as gaining a reference to a stage via discovery, a developer may create a reference to a stage manually using a network address or hostname. Listing 3.18 has an example of creating a reference to a stage at a known network address, in this case an IP address, and a hostname. Creating a reference to a stage in this manner does not guarantee the existence of a stage at the specified network address. This assertion is only tested when trying to use a stage in an adaptation operation, as discussed in the remainder of this section. This option can be useful when the topology of the distributed system is known *a priori*, and the ad-hoc discovery mechanism is superfluous.

Once a reference to a stage is obtained, it can be either sent to another actor or used to modify the runtime configuration of actors, as described in the following section.

3.4.4 Actor Adaptation

The motivation to adapt or change the runtime environment of different types of application in different domains has been established in Section 2.2. Consequently, this section will describe how Ensemble supports adaptation or reconfiguration at runtime from *within* the language.

Spawn

Spawning an actor is the process of creating a new actor at an explicitly specified `stage`; this stage can be the current stage, or a different stage. Spawning a new actor is similar to creating an actor with the `new` keyword, with the addition of a reference to the stage at which it should be created. As with `new`, the constructor of the desired actor should be supplied; this includes constructors which require any values or variables. Listing 3.19 shows different variations of actors being spawned.

```
1  behaviour{
2    stages = findStages(example_query);
3    if(stages.length > 0) then{
4      // create a baby actor without a reference
5      spawn baby() at stages[0];
6
7      spawn baby(1, "hello", example_proc()) at stages[0];
8
9      // create a baby actor with a reference
10     reference = spawn baby() at stages[0];
11   }
12 }
```

Listing 3.19: Spawn in Ensemble

```
behaviour{
  stages = findStages(example_query);
  if(stages.length > 0) then{
  // migrate the actor to the stage referenced by stages[0]
    migrate stages[0];
  }
}
```

Listing 3.20: Migrate in Ensemble

Actors may be spawned either with or without a reference to the new actor, Listing 3.19 lines 10 and 5, respectively. Actors which are spawned with a reference are semantically equivalent to an actor created with the `new` keyword, the only difference is that actors created with `new` can only be local, whereas actors created with `spawn` can be either local or remote. It is also important to note that actors which represent OpenCL kernels (Section 3.3) can also be spawned.

Should a stage be unreachable while trying to spawn, the `StageNotFoundException` will be thrown. If the actor cannot be created at the stage, an appropriate exception will be thrown in the actor invoking the spawn to reflect the problem. The most common exception is the `ConnectionFailureException` exception. If an exception is generated, the actor will not be spawned.

Migration

Unlike other actor-based languages, Ensemble natively supports actor migration without the need for extra-language constructs. Migration is an explicit action in the language which pauses the execution of the invoking actor, moves the actor, its channels, and its state to

```
behaviour{
  // replace the actor referenced by baby
  // with a new child actor
  replace baby with child();
}
```

Listing 3.21: Replacement in Ensemble

a different stage, and continues the execution of the actor immediately after the migration operation, Listing 3.20.

Migration is an actor specific operation, and can only be invoked by the calling actor; one actor cannot explicitly tell another actor to migrate. This follows the spirit of actor encapsulation and again serves to simplify the programming model. This also removes the need to have language primitives which will `fix` or `unfix` an actor to a particular stage, as in Emerald [109], which was required to prevent actors migrating each other. Such functionality could be provided at the application level if required.

Actors who have references to files or other resources which are located on a particular physical machine are not allowed to be migrated. This is enforced at compiletime as the compiler is aware of the relevant types. This limitation can be overcome by using actors and channels to abstract such location-dependent types. An example of this is seen in the media player application, Section 5.3.1. Here, file access is abstracted by an actor which accepts file names and returns byte streams. This approach could be extended by having all I/O interaction being abstracted by system actors. This would have the added advantage that such actors can be discovered at runtime. Currently, there are a number of system actors which abstract hardware access, however direct access is left to provide the developer with choice.

As well as all of an actor's code and state, any actor connected by channels before it performs a `migration` operation, will still be connected after a migration from either actor's perspective. Should an error occur during migration, the actor will not be moved to the new stage, instead continuing on the original stage, and an appropriate exception will be generated. This is a similar situation to spawning a new actor. Note that it is legal to migrate an actor to the stage it is currently on, however it is not legal to migrate an actor which represents an OpenCL kernel, this is discussed in Section 4.5.3.

Replacement

As well as the ability to move actors between stages, it was also desired that actors be *replaceable* at runtime. This is useful in situations where an actor is not operating as desired due to incorrect or outdated software. This situation is particularly observed in WSNs, where hardware is often difficult to access after deployment [110]. In such situations, it is preferable to update remotely via software, rather than retrieve the device, reprogram, and then re-deploy; this is generally known as over-the-air programming. Although not yet fully implemented, the following is a description of the semantics of replacement.

The replacement statement requires a reference to the actor to be replaced, and the constructor of the new actor, as shown in Listing 3.21. As with spawn, constructors with arguments may be used. The new actor will inherit all the channels and connections of the actor it replaces. Consequently, the new actor must `present` at least the same number and type of channels as the actor being replaced. This assertion can be partially guaranteed by the compiler at compiletime, as the channels of the new and old actor can be compared; if they do not match an error is generated. However, actors which are located via the discovery mechanism may present more channels than are specified in the interface which is used to locate them (Section 3.4.3). In this case, it is not possible to ensure correctness at compiletime. Hence, it is the responsibility of the runtime to ensure correctness. Should the new actor not be compatible, the `ActorNotCompatibleException` will be generated. As the language is designed to be used by non-experts or non-computing scientists, the use of reflection or runtime analysis to either indicate why the actors are incompatible or fix the issue was undesirable. Instead, a new way of discovering actors was introduced: `findReplaceable()`. This function is identical to `findActors()` except that only actors where **all** of the specified and examined channels match are considered as being eligible to be returned. In this way, `findReplaceable()` provides a mechanism to avoid the `ActorNotCompatibleException`.

While ensuring that the same channels are present in both the new and old actors provides type-safety and functional correctness, it does not guarantee that correct and safe interaction with other actors. This is especially true as the new actor does not need to have the same state, procedure or query definitions. To minimise this, an actor is conceptually replaced at the beginning of an iteration of its behaviour clause. This means that an actor must wait until all internal operations and external interactions with other actors have completed before being replaced. This provides a clear point at which to reason about the logic of replacement. Languages like Erlang and Scala support the idea of runtime code replacement, or "*hot-swapping*" of code, however this refers to logic, rather than the combination of logic and state.

Currently, the `replace` statement and the `findReplaceable()` operation are supported, hence there are no modifications required to the language to enable replacement. Also, the VM supports the discovery of replaceable actors, however, the final implementation of replacing an actor is not yet complete.

3.4.5 Location Transparency Via Channels

The use of message-passing is one of the main advantages of the actor model. Apart from facilitating the isolation of actors, it can also abstract the locations of actors while enabling communication. Hence, actors in the same or different stages can interact in the same way, without any change to existing code; all channel operations in Ensemble are location transparent.

Given that all distributed environments are subject to non-deterministic failures, this must be reflected within the language and is done so via exceptions. Exceptions related to communication in Ensemble are mainly generated from failure in the underlying communication medium or runtime.

As well as these failures, there is one which is generated from an inconsistent state in the language. The `ChannelNotFoundException` is generated by the language when a channel cannot be located during a `connect` operation. This situation can occur in between gaining an actor reference, either via a channel or discovery, and the referenced actor having migrated to a different stage or expired. This is an issue with the runtime as it only updates references with explicit connections, such as channels, on migration. This particular exception indicates that the last known stage was contacted and replied, but the desired actor and channel could not be found, as opposed to some hardware or software failure. In this case, a new reference to the actor must be acquired. This can happen for both local and remote channels, however, most often occurs for remote channels. This exceptions tells the user that the last known stage for the referenced actor is alive, and responding to queries. Should the stage be dead, the `StageNotFoundException` is thrown. In this situation the actor attempting to connect may either try a different channel or repeat the discovery process to locate the actor at its new stage.

Once a connection has been made between two channels, it will persist regardless if either of the connected channels is sent as data across channels, or the actor who owns the channels migrates between stages. It is the responsibility of the runtime to ensure this.

When using channels to communicate, the `TIMEOUT` exception is thrown to indicate that the attempt to push or pull data across the specified channel has gone beyond a system-defined timeout value. It is important to understand the situations in which this can happen, as a single unreachable connection is not enough to generate an exception. When attempting to push or pull data on a channel, a timeout will only be generated once *all* possible connections have failed to adequately indicate that they cannot convey data. For example, an exception will not be generated if at least one connection is able to indicate that it has no data available to push/pull, as the actor would block on this channel. This obeys the communication semantics discussed in Section 3.1.2, as the non-responsive channels can be considered as unconnected. An exception would only be generated if *all* connections do not reply with any useful information, which indicates a hardware or software failure.

Although an actor can only be discovered when published, once the channels of an actor have been bound to, the visibility of the actor has no effect on the ability to use these channels. Consequently, by discovering actor references through the discovery mechanism, the channels of an actor can be accessed and bound to; the location of the actor is irrelevant.

From a language perspective, location transparency is a simple goal to achieve as, by design, the channel operations do not change for local or remote connections. The main effort to support location transparency is found in the implementation, Section 4.4.2.

3.4.6 Stage Adaptation

The combination of reconfigurable channel topologies and actor placement enables the fine-grained modification of Ensemble applications at runtime. However, in some situations it is more desirable to operate at the more coarse-grained level of applications, rather than that of actors. The `stage` provides an appropriate language construct for this level of abstraction. Although stage migration is not currently supported, the following is a discussion of using the stage as a unit of application adaptation.

By using a stage to contain all actors of an application, or all actors of a certain equivalence class, and then migrating this stage between physical machines, entire sets of actors can be moved in a simple, logical action. From the actor's perspective they are still *acting* within the same stage, and all channel connections would be unaffected. There are a number of considerations with this approach.

Firstly, with this approach the responsibility is on the programmer to use stages in an appropriate manner. Just as a developer may use an actor per application or an actor per procedure, all actors could be held in a single stage, or spread across multiple stages.

Secondly, migrating an application of actors is currently possible without migrating stages. By maintaining a set of channels connected to actors in an application, a user could manually inform each actor that it should migrate to a specific stage, with each actor listening for this message and migrating itself. Here, an array of channels can be used to abstract an application of actors. This has the advantage that the actors need not be on a single stage, but does require the user to manually inform each actor to migrate. Stage migration would offer a simpler abstraction.

Thirdly, stages are currently created and destroyed outwith the language, from the command-line. Although the ability to move a stage between physical locations is not yet implemented, stage migration would also be controlled in the same manner, and would not require any modification to the language. Manipulation of a stage in this manner would enable a third party to control load balancing of sets of actors based on external factors, such as resource constraints or policy decisions. Similarly, implementing load-balancing within the runtime would enable automated stage/actor reconfiguration, but this is an area of future work.

By enabling adaptation at both the actor level and the stage level it would be possible to have both coarse-grained and fine-grained control over the runtime arrangement of loci of computation and resource consumption in a way which is natural to the actor-model.

3.5 Summary

This chapter introduced the Ensemble programming language which was created to enable an exploration of the hypothesis with respect to actors, given the limitation of other existing approaches as identified in Section 2.1. The use of actors naturally provides race-free memory access, and parallelism. In addition, the following three areas have been introduced.

Memory Consumption To reduce runtime memory consumption required by the shared-nothing semantics of the actor model, a *movable* memory space was added to the language. Unlike similar approaches, movability is expressed very simply in the language by a single keyword (`mov`), using compiletime analysis to ensure its correct usage.

Accelerator-Based Concurrency Noting the parallel between accelerator-based concurrency and the actor model of programming, Ensemble is the first to natively enable an OpenCL kernel to be expressed as an actor, where the idea of host and device interaction is more naturally expressed as actors, and explicit memory movement is abstracted by channel operations. Memory optimisations are available through the use of movability. Also, by expressing the kernel as Ensemble source code, compiletime assistance is provided to create correct kernels, and verbose initialisation of OpenCL is automatically handled by the runtime.

Adaptive Programming By leveraging the encapsulation of actors, the remote creation, relocation and replacement of actors at stages is provided natively within the language. A language mechanism is provided to both describe and discover actors or stages based on inherent properties, such as type, as well as user-defined properties at runtime, rather than predefined configuration files. Furthermore, channel connections between actors are supported both locally and remotely. This is irrespective of the physical location of the actor or stage.

Chapter 4

The Ensemble Virtual Machine

In order to realise the actor-based programming model presented by Ensemble in Chapter 3, it was necessary to create a runtime environment. This runtime needs to support the actor-based programming model, as well as runtime adaptation, location transparent channel operations, and OpenCL kernels. Additionally, this must be done in a way which is supported across multiple hardware platforms with vastly different operating characteristics and resource constraints.

To achieve this, it was decided that Ensemble applications would be compiled into an IR and executed by a VM. Having a platform-independent representation enables the adaptation described in the language to be achieved without the need for multiple compiler backends and hardware specific binaries. Instead, the compiler need only target a single VM. Section 2.2 discusses the merits of different intermediate representations. Java bytecodes were chosen as the instructions for the VM due to general familiarity Java bytecodes in the computing community. Howeverm these bytecodes have been modified to meet the size and memory requirements of this work. In this section the term runtime will be used to refer to the combination of the VM and the C-based InceOS implementation upon which it rests. InceOS is described in Section 4.1.

Section 4.2 describes how Ensemble applications are compiled to augmented Java bytecodes. Section 4.3 discusses the design and implementation of a modified JVM to execute Ensemble applications. This work is expanded upon in Section 4.4 to include support for runtime adaptation and inter-stage channel communication. Support for the abstraction of OpenCL kernels by actors is discussed in Section 4.5. Section 4.6 describes the modification to the compilation process and runtime to support Ensemble applications on embedded hardware. There is a brief discussion of security concerns caused by the actor model supported in this work in Section 4.7, and then a summary of the points raised in Section 4.8.

4.1 InceOS

The virtual machine described in this chapter is built upon the actor-specific operating system InceOS [22]. InceOS was originally designed to execute on the embedded Tmote Sky [111] platform, however, during the course of this work, it has been expanded upon and ported to more platforms. A more complete description of the internal mechanics of InceOS, as well as a detailed space and performance evaluation has been documented [112].

InceOS is a C-based pre-emptive multitasking operating system, where the unit of execution is the actor. By default, each actor is represented as a data structure containing system information, the actor's state and channels, plus a single thread of control. Section 4.3.2 describes the use of threads in more detail. The OS provides a system call API to control the life cycle of actors: creation, modification, and destruction. The OS also natively supports the interaction of actors via message passing, ensuring the shared-nothing semantics. Again, this control is provided via a system call API.

The OS enables dynamic memory allocation using either first fit or best fit allocations policies, however, this can be configured and/or extended. Also, the OS provides garbage collection via a reference counting system. Access to system hardware, such as timers, radios, file systems, networking, and sensors is achieved via system actors, discussed in Section 3.1.8. This helps to remove environmental dependencies by having user actors interact with system hardware via message passing, thus giving location transparency.

The default scheduling of actors is done in a round-robin fashion, where system actors which have been woken by a hardware interrupt, such as a button press, are scheduled at the head of the queue. As well as facilitating message communication, the OS is able to use the channel-based connections between actors to help schedule the actors. Specifically, the OS only contains a run queue and a kill queue. Actors which are blocked waiting for messages do not need to be stored in a queue, and are only rescheduled when a message is either being requested or delivered. This is achieved by examining the references which are held internally by the OS. Hence, the interaction of actors determines their scheduling. Should actors be computationally intensive and not block, they will be pre-empted, and the system will default to round-robin to ensure fairness.

4.2 Compiling Ensemble Applications

Figure 4.1 shows the process of compiling an Ensemble application to an executable format. Firstly, the Ensemble source is translated to Java source code. Section 2.2.3 discusses the justification of using Java bytecode. Next, the Java source is compiled to Java bytecode using the Java compiler (`javac`). The class files which are generated are combined with

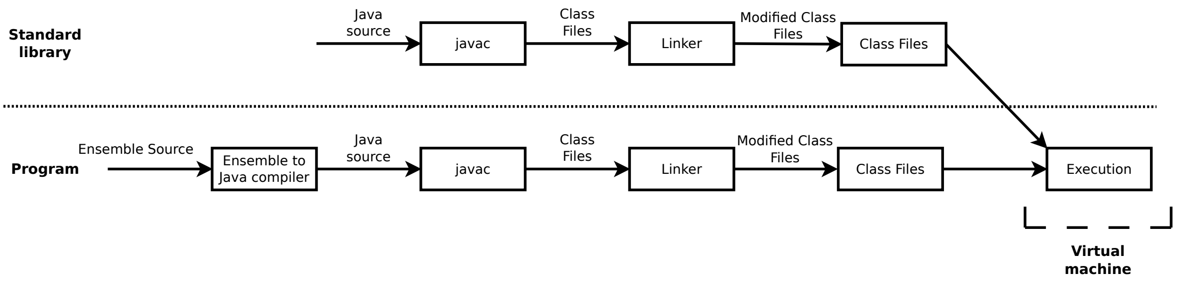


Figure 4.1: Ensemble Compilation Process

```
ChannelIn<integer> int_chan = new ChannelIn<integer>(0, "i");
```

Listing 4.1: Java Definition of an `IN` Channel with no Buffer, which Conveys an Integer

a standard library, also implemented in Java, and are then passed through a custom *linker*, which generates new class files containing augmented bytecode. These bytecodes are then executed by the Ensemble VM.

4.2.1 Representing Ensemble In Java

After the validity of an Ensemble application has been checked by the compiler, it generates a Java source code version of the application. This is straightforward as Ensemble's arithmetic operations, memory allocation, and array or structure dereferencing are naturally expressed in Java. All actions which are specific to Ensemble, such as actor, channel, discovery, adaptation or OpenCL operations, are represented by Java methods. As these are operations, Java classes are not required. These methods are defined in a standard library, but only act as wrappers to underlying operations in the runtime, see Section 4.2.2. The actors, channels, interfaces, structures, and arrays are represented as Java classes.

Actors

Each actor in an Ensemble application is compiled to a Java class which extends a standard `Actor` class. Both actor state and channels from interfaces are stored as fields in the class. The predefined `Actor` class is necessary to provide a static `start` method which is used to begin the execution of an actor as a new and independent entity. There is also a static `stop` method, which is used to stop the execution of the actor. Both functions act as hooks into the VM.

Channels

A channel is represented by either the `ChannelIn` or `ChannelOut` class, where one class is instantiated for each Ensemble channel. The classes are generic, meaning that the data type which is conveyed by the channel is generated by the compiler, rather than having all channels convey the `Object` type. `javac` will type-check all channel operations at compiletime in addition to the Ensemble compiler, hence no runtime type-checking is necessary. The Ensemble compiler outputs the appropriate type of the channel based on its definition in the application. Listing 4.1 shows the Java representation of a channel which accepts an integer. The constructor for the channel will take an integer to indicate the size of the channel's buffer and a string representing an encoding of the type of the data conveyed by the channel. The string encoding is needed for remote channels, and is discussed in Section 4.4.2. Note that only `in` channels may have a buffer specified, as described in Section 3.1.2.

Channels are one of the classes treated specially by the VM. A Java channel wraps a native InceOS channel, and applications use native Java methods to access the underlying InceOS C functions. These functions pass data opaquely from sender to receiver, so the size of the type being passed must be specified when creating an InceOS channel directly. An advantage of generics is that all Java channels simply pass a pointer, hence the specified size is the same for all data. A drawback to using generics is that primitives must be *boxed* and *unboxed* before and after being passed over a channel, which can be time-consuming if a channel is used frequently.

Structures

Structures defined in Ensemble are also represented by Java classes, where the fields of the Ensemble struct are replicated in the Java class. Ensemble interfaces are defined similarly, but only contain channels. In this context, an Ensemble interface has no relation to a Java interface. Both structures and interfaces extend the `EnsembleStruct` class which contains the prototypes of the `duplicate()` and `onReceived()` functions.

The `duplicate` method is used to create a deep copy of the struct and its fields before it is sent across a channel. For each struct class, the compiler will generate appropriate Java code to duplicate the fields used in the class. The `onReceived` method is invoked on the data received from a channel. This is necessary when the data being sent is a channel, as the ownership of the channel must be transferred to the receiving actor to maintain actor encapsulation, see Section 4.3.4. Again, the compiler will generate appropriate code to override this function if required.

As an aside, Ensemble variables of type `any` are represented by the `Object` type, with other objects being cast appropriately.

4.2.2 Linker

After the Java source code has been generated, both it and the standard library are compiled by the Java compiler to generate class files. The resulting class files are then passed through the Ensemble linker. The purpose of the linker is to modify both the structure and contents of the class files to be more suitable for Ensemble applications.

Java bytecodes were chosen as the intermediate representation due to the maturity of both Java and its community. Specifically, there are many tools and libraries designed to work with Java bytecodes for development, optimisation, and analysis. Also, Java bytecodes and the JVM are well documented. This serves to simplify both current and future development of the Ensemble VM.

Interaction with Native Code

The Ensemble VM is built upon InceOS directly, meaning that the lifecycle operations of actors and channels are implemented in C. In Java, this functionality is accessed through *native* Java methods. A native method is called from the JVM [113], but has been implemented in a different language. This necessitates the use of a C-like stack, in addition to the JVM thread's stack. In a standard JVM, native methods are called using the same instructions as normal methods, and the constant pool is used to resolve and load the necessary native code.

Instead of this, a new `invokenative` bytecode instruction has been introduced within the Ensemble VM specifically for calling native methods. This instruction behaves like `invokestatic`, except that its argument is treated as identifying a native method rather than an interpreted method. The structure of most native methods is to extract native data from the arguments, to make a system call, and to convert any result back into Java form. The existing invocation instructions `invokestatic`, `invokespecial`, and `invokevirtual` are only used to invoke interpreted methods.

A standard library of Java native methods is used to represent certain InceOS operations, such as sending data across a channel. These methods are only wrappers for native methods, and are declared with the `native` keyword to indicate this to the compiler. The linker will generate a C header file containing both a macro per wrapper function and the number of stack frames used by that function's parameters. This is necessary to clear the parameters from the stack after the function has been called. This approach means that native methods must be static. There is currently no support for virtual invocation of native methods, but this has not been found necessary in implementing Ensemble applications.

The new bytecode has reduced the amount of space required at runtime as no bytecode representation of the standard library is required. Instead, the interpreter need only use the number of the native method, as defined in the macros of the generated header file. This is more efficient as most native methods, such as creating channels, will always be present within the runtime. For those which are not, conditional compilation can be used to reduce runtime resource consumption. This is particularly useful when targeting embedded platforms, see Section 4.6.

4.2.3 Dependencies

Although actors do not share state, they do share types. This is true in both the language and the runtime, where it is necessary to have actors create user data types or invoke procedures. While this sharing does not lead to inconsistent state in the language, it does generate dependencies between the class files which represent these types at runtime. This is necessary for actor adaptation, see Section 4.4.3.

To address this, the compiler determines the dependencies between the different Ensemble entities before generating the Java source code. When creating the Java classes that represent the Ensemble types, the compiler generates the custom `@dependency` annotation for the class being generated. This annotation contains a list of classes upon which the class currently being generated depends. The linker uses this annotation, and embeds both the number and names of the dependencies into the class file for the particular type. During any action which would potentially require a new data type to be present in a stage, such as a spawn or migrate, the runtime can use this information to determine any unmet dependencies. In the future, it would be better to use runtime analysis of the class file's symbolic references to determine dependencies, rather than store these directly in the class file, thus reducing the size of class files.

As well as explicit inter-class dependencies, the compiler also optimises for subroutine usage. Procedures or queries which are defined either outwith or within an actor only have Java code generated if they are invoked within an actor. This means that space is not consumed for subroutines which are never invoked. However, this does mean that each actor will have its own copy of any invoked subroutine. Although this may lead to multiple copies of the same function, it is useful for reducing inter-class dependencies, as well as simplifying the spawn and migration process discussed in Section 4.4.3.

Class File Format

Java class files contain member definitions (fields and methods), metadata, and constant pools. Past analyses of Java programs show that, on average, class files can be as little as 33% method definitions [114], and only 20% bytecode [115]. However, this may not

Data	Standard library (33 class files)	Programs (20 class files)
Constant pool	66.5%	67.5%
Class metadata used	0.2%	0.2%
Class metadata unused	5.2%	7.6%
Field metadata used	0.2%	0.2%
Field metadata unused	7.4%	1.5%
Method metadata used	2.3%	2.7%
Bytecode	3.5%	7.3%
Method metadata unused	14.6%	13.0%
Method total	20.4%	23.0%
Total used	72.8%	78.0%
Total unused	27.2%	22.0%

Table 4.1: Average percentage composition of class files. 'Used' and 'unused' indicate whether information is present in the modified class file.

be representative of Ensemble applications, and in particular the standard library, which contains mainly class and native method definitions.

The class files from the standard library and various Ensemble programs have been analysed to find how much of their data can be discarded. The results are shown in Table 4.1. Most of the unused data is related to linking. The rest is mainly metadata related to Java features unsupported by the VM, or which is encoded in the VM's new instructions (e.g. the size of fields, and whether methods are native). Appendix B fully describes the new class file format.

Inter-Class References

Currently, the class files representing specific Ensemble entities are symbolically referenced by name. If all Ensemble applications were compiled from a single source file, the compiler would be able to ensure that no two types could have the same name, hence this referencing approach would be safe. However, as Ensemble applications are designed to be able to work together when compiled independently, this approach does not guarantee uniqueness - two distinct types may posses the same name.

To solve this problem using a decentralised approach, a unique naming scheme is adopted. By taking a MD5 hash [116] of an Ensemble type's class file at compiletime, a 128-bit identifier is produced to identify a class in place of a literal name. The Java Universally Unique ID (UUID) library[1] is used to generate this number. Using a hash of the post-linked class file has the advantage that if two identical actors are compiled independently, they will

[1]http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html - Accessed February 2015

posses the same UUID. The generation and use of UUIDs is not visible to either the language or the user.

As a UUID is represented as a finite number, it does not guarantee a truly unique number - it is possible that two different classes will hash to the same UUID. There are a number of points to consider. Firstly, by inspecting the number of bits used in the UUID and the approach used in the library, there are 2^{126} potential values which the UUID may take. This is sufficiently large for the purposes of this work, making a collision extremely improbable. Should this not be sufficient, it may be possible to increase the number of bits used to represent the key. Also, metadata may be used to add context specific information. Examples of such meta data include the literal name of the type, or the string encoding used for the type. Secondly, as the UUID is generated from the class file, it is possible that two actors which have been named distinctly may be found as equivalent. This is essentially structural matching. Unlike the discussion in Section 3.4.3, this will not lead to unexpected logical errors. For raw types, there is no issue in choosing one type over another if they are structurally the same. For actor classes, the UUID includes the actual implementation of the behaviour clause in addition to the data types used. This means that both the actor's state and logic are used to generate the UUID, hence the UUID is generated from a unique representation of the actor.

Encoding

Although the compiler ensures that only valid Ensemble applications will compile successfully, the presence of runtime discovery, reconfiguration, distribution, and the `any` type requires that there exist some encoding of an entity's type at runtime.

Encodings fall into two categories: those which represent primitive types, and those which represent aggregate types, such as structs and actors. Table B.1 in Appendix B describes the mapping between types and encodings. This string-based encoding was chosen as it was simple, both to implement and to perform type comparisons at runtime. In the future, using the hash of an encoding, rather than the encoding itself, may be more space and time efficient.

Similarly to the inter-class dependencies discussed in Section 4.2.3, the compiler will determine an encoding for an entity's type at compiletime, and annotate the class file generated for this type with the custom `@encoding` annotation containing the string encoding. The linker will then encode this information into the class file. Also, an encoding of the data which a channel conveys is supplied to a channel when created. This is necessary for communication across remote channels, see Section 4.4.2. As runtime type information is only required in the distributed case, its use has been kept to a minimum.

Optimisations

Although Ensemble applications are translated to Java source code, and then compiled using the Java compiler, they do not use all the features of Java. As well as not implementing a number of Java features, the linker performs a number of optimisations to reduce the size of the class file.

'Static Only' Classes In Java, all methods and fields must be part of a class; there is no such thing as a 'top-level method' or a global variable. However, some classes within the standard library contain only static methods and static fields which are never instantiated, never extended, and never used as the type of a variable. As this only applies to classes which are integral to the runtime and are always present on each stage, there is no need to include a class definition for them; only the methods and fields themselves are included. A class suitable for this treatment is marked with the custom `@static_only` annotation by the compiler, which is detected by the linker. All wrapper classes for system actors are marked as static only, giving significant space savings.

Empty Methods Java object construction occurs in two stages. The `new` instruction allocates the memory required for an object, and then the object's constructor is called. All Java classes are required to have a constructor. `javac` generates default constructors where necessary, which do nothing but call the superclass's constructor, to ensure this rule is satisfied. This leads to long chains of calls to methods that do no useful work at all.

The linker detects these methods and removes them from the generated class files. All calls to these methods are also removed. This continues iteratively until all constructors and static methods which do nothing, transitively, have been removed. Some modification of the bytecode around the removed call sites is necessary to ensure correct execution. In particular, method arguments which are pushed to the operand stack before a call must now be disposed of as the call no longer happens. If an argument was pushed immediately before the call, the pushing instruction is removed; otherwise, an appropriate number of `pop` instructions are inserted.

Virtual methods are not removed even if they do nothing, so that virtual calls continue to work as expected.

Native Methods The `invokenative` instruction refers to native methods by IDs assigned by the linker, made available to the VM through a C header file. Hence no information about native methods is stored.

Direct Bytecode Generation Ensemble applications are currently compiled to Java, and then to bytecode using `javac`, before being modified by the linker. There would be several advantages in compiling Ensemble directly to augmented bytecode:

- *Reduction in code size* – a number of the classes and methods in the standard library exist only for compatibility with the code generated by `javac`. Certain methods in the 'primitive classes' such as `Integer`, and classes to support exceptions, are not strictly necessary, but are required to link with classes generated by `javac`.
- *Optimisation of bytecode* – the code generated by `javac` might not represent Ensemble idioms in the most efficient way. Optimising at the compilation stage, with knowledge of the changes to the instruction set made by the VM, might be easier than attempting to optimise `javac`-generated bytecode in the linker.
- *Variable and stack usage* – the Ensemble VM uses bitmaps to record which local variables and stack slots contain objects, so that garbage collection works correctly. These must be maintained at runtime because `javac`-generated code reuses slots for different types throughout the lifetime of a method call. By contrast, in Darjeeling (Section 2.2.2) a frame has separate variables and stacks for objects and primitives, so that the information needed for garbage collection is available statically [49]. This is possible because of the extensive bytecode rewriting Darjeeling performs during the linking stage. A similar process could be adopted in Ensemble.

Although there are clear advantages, time constraints prevented the use of direct bytecode generation in this work.

4.3 Ensemble VM Design and Implementation

Once compiled to Ensemble class files containing augmented Java bytecodes, Ensemble applications are interpreted by the Ensemble Virtual Machine (EVM). The EVM is a specialised Java virtual machine [113] which is designed to execute Ensemble applications on a range of hardware platforms including highly resource-constrained platforms, reasonably-provisioned platforms, well-provisioned platforms, and highly-parallel hardware platforms. The EVM is implemented on top of InceOS both on bare metal and on Linux-based platforms.

The choice to create a new VM, as opposed to using the JVM, was three-fold. Firstly, the JVM is designed to execute Java applications. As described throughout Section 4.2, there are a number of Java features which are not useful to actor-based applications, hence the JVM does not best fit the needs of this work. Secondly, although targeting the JVM would enable Ensemble applications to execute on a large number of JVM supported hardware platforms, these platforms predominately fit into the reasonably to well-provisioned category. This work is focused on exploring the use of actors in a wider spectrum of hardware platforms. Also, as the hardware platforms become more constrained, so too does the support for Java features. Thirdly, targeting the JVM would not enable the native support of actor runtime adaptation, specifically thread migration. To do so would require the JVM to enable access to the state of running threads, which will never be supported in the JVM due to security concerns. As discussed in Section 2.2.5, the alternatives either increase space overheads, increase performance cost, or require the use of modified JVMs.

As the EVM is intended to operate across a large number of hardware platforms, a balance must be found between space consumption and performance. For smaller, resource-constrained platforms, the EVM must optimise for space, whereas, the EVM for larger platforms must optimise for time. Hence, the following discussion will leave the reader noting a number of obvious optimisations.

4.3.1 JVM Support

Although executing a form of Java bytecodes, the EVM is not designed to support Java applications. This is in contrast to other actor languages such as Scala and Salsa which are compiled to Java bytecodes in order to run on JVMs. Consequently, the following JVM features are not supported:

- *Synchronisation* - As there is no shared state or locking mechanisms in Ensemble, there is no need to `synchronise`. More generally, there is no need to support the Java concurrency model.
- *The Java Standard Library* - Ensemble has its own standard library. Only a minimum number of classes from `java.lang` are supported, with as few methods as possible. These classes include `Object`, `String` and primitive wrapper classes. They are necessary due to the use of Java source code in the compilation process.
- *Reflection* - Ensemble does not enable the use of reflection in applications. Apart from the complexity that this would introduce, it also has the possibility to break actor encapsulation. Consequently, no API support is available for reflection. Although some type information is available at runtime for use with adaptation, it is only visible to the runtime.

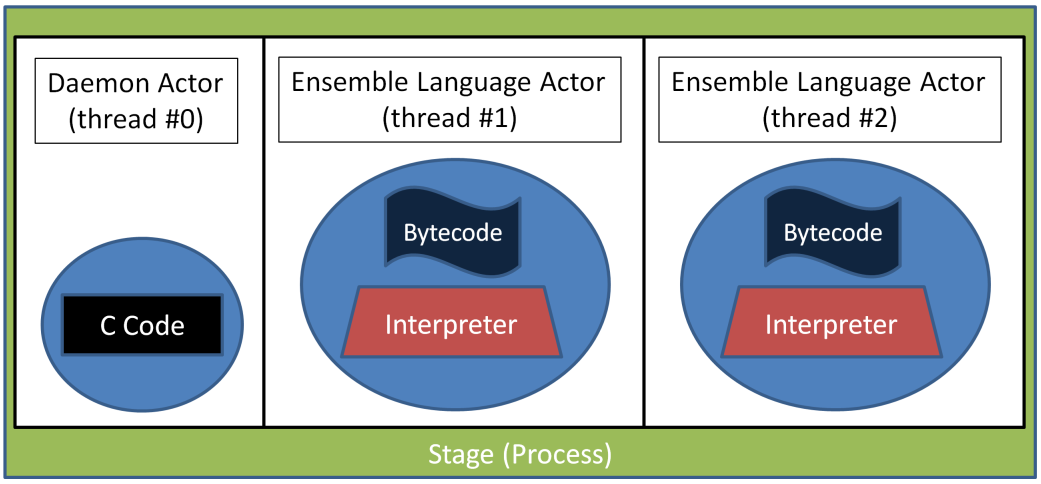


Figure 4.2: Architecture of the Ensemble VM

- *Interfaces* - Ensemble does have an interface type, which is used to add channels to actors. These interfaces are distinct from Java interfaces. As Ensemble is not object-orientated, interfaces are not used to dictate the implementation of actors in the Java sense. Although inheritance is used in the Java representation of an Ensemble application, this is done for convenience and is not visible to the developer.

As these features are not used, the bytecodes associated with them have not been implemented. Some features, such as multidimensional arrays and exceptions are supported, but in a restricted fashion.

4.3.2 Structure of the Ensemble VM

The structure of the Ensemble VM is shown in Figure 4.2. Each VM represents a `stage`. A stage is described as a memory space, rather than a physical machine, because multiple stages may conceptually exist within a single machine. Note that this is not currently the case, see Section 4.4.2.

The VM either executes as a process on Linux-based systems, or as the sole application on InceOS-based systems. Within the VM, a thread is created for each actor. When the VM is executing on Linux-based systems, the `Pthread` library [117] is used. When the runtime is executing on InceOS, the InceOS thread library is used. Each actor thread executes C code which interprets the bytecode representation of an actor's behaviour clause. The thread executes the interpreter in an infinite loop until explicitly told to `stop` in the language, at which point the actor and its resources are garbage collected, and the thread exits. The daemon actor is implemented in C, and is executed natively by a thread, rather than an interpreter.

Platform	RAM	Storage	Processor	OS	Networking
Desktop	16GB	1TB	Core i7: 4 cores @ 3.3GHz	Linux	Ethernet/wifi/Bluetooth
Laptop	8GB	500GB	Core i7: 2 cores @ 2.4GHz	Linux	Ethernet/wifi/Bluetooth
GPU	4GB	N/A	64 threads x 44 cores @ 1.03GHz	Linux	N/A
RaspberryPi	256MB	8GB	1 core @ 700MHz	Linux	Ethernet/wifi/Bluetooth
Lego NXT	64KB	256KB	1 core @ 60MHz	InceOS	Bluetooth
Tmote Sky	10KB	48KB	1 core @ 8MHZ	InceOS	Zigbee

Table 4.2: Specification of the Ensemble Supported Platforms

The VM maintains lists of the loaded classes. It also keeps lists of all created actors and channels.

4.3.3 Supported Platforms

One of the key challenges being addressed by this work is the growing use of connected, heterogeneous hardware platforms, and the challenge of programming such systems. In order to show that applications using the actor model simplify programming such systems, it was necessary to implement the EVM on a number of different hardware platforms. There are currently seven supported hardware platforms, which are described in Table 4.2. The platforms have been split into four equivalence classes which represent a range of different scales of computing hardware:

- Highly-provisioned and parallel hardware in blue.
- Normally-provisioned or common hardware in green.
- Reasonably-provisioned hardware in orange.
- Highly-constrained or embedded hardware in red.

These platforms were chosen to show both the applicability of the actor model of computation at different levels of computing scale, and also the feasibility and usefulness of adaptability across the platforms. Note that parallel hardware architectures, such as the GPU and multicore platforms, are accessed via OpenCL, which is discussed in Section 4.5. Also, due to severe resource constraints, the EVM is implemented in a reduced fashion on the Tmote Sky platform. This is discussed in Section 4.6. Because of this, the EVM was ported to the Lego NXT platform. Although it has more resources that the Tmote, it has substantially less than the RaspberryPi, and is an example of an embedded hardware device. Time constraints prevented an implementation on a mobile phone, but this is the next logical platform to support.

Excluding the Tmote Sky, each platform runs the same version of the VM (with appropriate hardware drivers). Hence, given sufficient space on the platform, any Ensemble actor may execute on any platform without requiring cross compilation, recompilation, or modification in any way. A complete discussion of the differences in the implementation for the Tmote Sky platform can be found in Section 4.6, however, the main differences are that Ensemble applications on this platform are statically compiled to a single binary, and do not support inter-node adaptation. The embedded version does run Ensemble applications on the EVM, but does not support location-transparent communication via channels, or the runtime creation or relocation of actors to different hardware platforms, although explicit inter-node communication is supported.

4.3.4 Communication Model

In order to implement the communication model in the language, the EVM uses the functionality provided in InceOS which natively supports both actors, channels and their respective operations. Example of these operations include `send`, `receive`, `stop`, and `migrate`. As discussed in Section 4.2.2, the operations on actors and channels are implemented as native Java methods which are modified by the linker to be custom `invokenative` bytecodes, with the argument being the operation to perform. During interpretation, the EVM will use this argument to call the relevant InceOS function, converting any arguments on the Java stack to parameters to the C function. As these functions are well-defined, the number, type, and order of arguments are known *a priori*. After the function has returned, the VM removes the relevant arguments from the stack, as well as converts any returned values from the C function and places them on the Java stack.

The functions which provide the channel operations mirror both the operations and semantics described in the language, see Section 3.1.2. Although there are no locking mechanisms in the language, the runtime requires their use to ensure serialised access to certain operations, especially data transmission. This is especially true on hardware platforms which support parallel execution. When running on bare metal, InceOS has complete control of interrupts and thread scheduling. Also, on these platforms there is only a single processing core, meaning that briefly disabling hardware interrupts is sufficient to guarantee serialised code execution. When using Pthreads, scheduling is more challenging. In this case, a single mutex is used to ensure serialised access to all channel code, as well as a single mutex for actor code. As channels and actors contain references to each other, and actors operate in parallel, two separate mutexs are required because some actor and channel operations require access to both data structures, and some only require access to a single data structure.

The use of a global lock simplifies the implementation and saves on space, but acts as a bottleneck because actors are fully isolated entities which are capable of executing concurrently. In the future, it is possible that for resource rich platforms, the VM could be implemented with a per actor/channel locking mechanism. Indeed, a formally verified proof already exists for the send and receive operations [118]. As stated at the beginning of this section, there are many ways to implement the language model.

Scheduling

An advantage of using blocking communication is that the interaction of actors via blocking channels naturally dictates the scheduling policy of the actors themselves. An actor which has blocked on a channel can only become eligible to run through the rendezvous of channel actions; the most recent rendezvous action unblocks the actor. This is possible as each channel is aware of the actors to which it is connected; thus when a channel action occurs, only the relevant actors are examined. For actors which are compute intensive, the runtime uses pre-emption to ensure execution fairness. Currently, user actors are scheduled in a round robin manner, with priority given to system actors. The exploration of different scheduling patterns or the use of priorities at runtime is left for future work.

Duplicating Data For Communication

As discussed in Section 3.1.2, data is normally duplicated when sent across channels in order to preserve the shared-nothing semantics of the actor model. When channels are sent between actors, it is not enough to duplicate the channel object to be sent. In addition to creating a new channel and replicating any existing connections to other channels, the new channel must be *adopted* by the receiving actor. Adoption is the process of adding a reference between an actor and a channel, as well as a reference between the channel and the actor. This bidirectional relationship is necessary for channel connections and communications, as well as actor scheduling: a channel must always be owned by an actor. The `onReceived()` function in both the `ChannelIn` and `ChannelOut` classes is used to execute adoption when a channel is received, otherwise it is invoked directly by the runtime.

4.3.5 Memory Model

The EVM is a stack-based VM, like the JVM. An alternative register-based model was rejected due to the large memory requirement, despite the better potential for optimisations [119]. By comparison, stack-based bytecode tends to be slower but smaller than equivalent register-based code. Using fewer resources is generally beneficial, however, the need to limit memory consumption was required for certain hardware platforms, Section 4.6.

Slot Size

To support the stack-based model, a new call frame or stack frame is allocated for each procedure call. Like the JVM, the Ensemble VM allocates a new frame from the heap as required, as opposed to using a static number of pre-allocated frames.

Every call stack frame has an operand stack consisting of fixed-size *slots*. Local variables are stored in separate slots of the same size. The JVM specification requires slots to be 32 bits, to match the word size of common desktop CPUs. Values of all data types occupy one slot, except for `long` and `double`, which occupy two. However, instructions are defined in terms of the number of slots upon which they operate, with no reference to the actual size of the slots. This means that the slot size can be changed without any change to the bytecode, as long as each data type still occupies the same number of slots. This is relevant for the discussion in Section 4.6.2.

Stack Frames

A minimal stack frame with no slots and no local variables occupies 32 bytes of RAM, with each additional slot or variable requiring four bytes. The number of slots and variables used by a method is known at linktime, so the whole frame can be allocated as a single unit. This has the advantage that only the memory currently required is used by the VM stack, rather than pre-allocating a stack based on the worst case need. Memory must be allocated in advance for an actor's C stack, which is used to run the interpreter and native methods. This is on the order of a few hundred bytes per actor when using InceOS, and is an internal default value when using the Pthread implementation on Linux.

A stack frame contains a pointer to the previous stack frame, the return address, a reference to the method being executed by the frame, and arrays for the operand stack and local variables. Additionally, stack frames contain bitmaps used to track which of the operand stack slots and local variables contain references; this is currently required for garbage collection

Objects

All objects are allocated on the heap. A class definition includes a reference to the class's superclass (as in standard Java, all classes descend ultimately from `Object`), the size of its fields, and a virtual method table.

When an object is instantiated, space is allocated for its fields. Unlike stack slots, fields can differ in size, and are packed in memory. The size of a field must therefore be known when accessing it. A standard JVM keeps this information in the constant pool, but the EVM instead uses new type-specific versions of the `getfield` and `putfield` instructions. These have been introduced for the different field sizes to reduce the amount of information contained in the classfile. The instruction to use in each case is chosen at linktime.

Some classes are treated specially. `String` contains a pointer to a native string. Arrays contain a pointer to a native array, as well as the size, dimensions, and element class of the array. The class of an array itself is the special placeholder *array* class, and variants of the `instanceof` and `checkcast` instructions have been introduced to test the element type and dimensionality of arrays. Additional space is allocated for these classes by the VM.

Static Fields

Ensemble does not support static fields as they could break the strict encapsulation of actors. They are not allowed in bytecode programs.

Garbage Collection

The VM uses the reference counting garbage collector provided by InceOS. All objects are reference counted. Bytecode instructions which manipulate objects also increment and decrement the reference counts appropriately. The choice to use reference counting is because the EVM is built upon InceOS which was designed for embedded systems, where the need to efficiently return memory to the heap as soon as possible is required. This said, the choice of garbage collection technique is orthogonal to the use of actors.

It is necessary to monitor, at runtime, which slots and local variables in a stack frame currently contain references, so that when a method returns, their reference counts can be decremented appropriately. This is done using bitmaps which are allocated along with the stack frame.

As with any basic reference counting system, the InceOS collector cannot handle cycles in the object graph. To some extent the Ensemble language mitigates this by always duplicating complex data types which are sent over channels, however, as structures may reference each other, cycles are possible. Also, should the user circumvent the language rules (e.g. by providing hand-written Java code to `javac`), then the system cannot guarantee that objects will be collected. The presence of cycles can be mitigated through the use of cycle detection [120] or the use of a tracing collector. This would either require a modification of the existing mechanism, or the implementation of a new garbage collector, respectively.

In the Ensemble VM, the use of reference counting influences how out-of-memory conditions are handled. In most reference counting systems, a tracing collector is present as a backup. This is run when there is not enough memory to service an allocation request, so that any cycles no longer needed can be collected. Only if there is still insufficient memory is an out-of-memory error signalled. In the Ensemble VM, however, no such backup collector is currently present. An out-of-memory condition results in an exception being thrown; if this is not handled, the actor is restarted. If the VM itself has insufficient memory to generate the error, it will fail. This is particularly problematic on embedded systems where failure is *hopefully* indicated by a flashing led.

Movability

The use of movability in Ensemble was primarily designed for highly resource-constrained platforms, as the increased heap usage and fragmentation can represent a non-trivial reduction in the amount of available RAM. Consequently, it was important that the correctness of movability be determined at compiletime, rather than runtime. As a result, the only manifestation of movability at runtime is that the compiler will not generate code to duplicate data allocated from the movable heap before being sent over channels.

Also, even though the language model describes two heap spaces, there is only a single heap from which all data is allocated. The compiletime analysis ensures that data from the two conceptual heaps will not interact. Movable channels sent between actors must still be adopted upon receipt; even without duplication, they must transfer ownership from the sender to the receiver.

4.4 Adaptability

This section describes how adaptability is implemented - specifically, the ability to discover actors and stages, use channels to communicate with other actors regardless of location, as well as the ability to spawn, migrate and replace actors at runtime. As noted previously, the shared-nothing semantics of actors, coupled with explicit message passing, presents the perfect computational model for distributed and adaptive computing. While other actor languages support spawning actors, this work is the first to natively support the strong migration of running actors between different types of hardware platform from within the language.

4.4.1 Discovery

The discovery of actors and stages at runtime is split into two parts: how these language types are identified and referenced, and how the runtime supports interacting with different platforms across different technologies.

The Query Type

As described in Section 3.4.2, queries are used to identify sets of actors or stages at runtime. To achieve this, the boolean expression within the query definition is converted into a bytecode representation at compiletime. As only primitive values may be used, the bytecode need only express simple arithmetic or boolean operations. The only exception is the `can_run()` operator, which is encoded as a bytecode itself. Hence, the interpreter for these operations is much simpler than the interpreter of the actor's behaviour.

Using the values and boolean operations from the query definition, a `query` instance is created at runtime. This cannot happen at compiletime as a query may use variables, the values of which are not known at compiletime. The compiler will generate a list of bytes representing the boolean operations, in the order which they should be executed. Also, the depth of the stack required to compute the query is calculated at compiletime, so that only the required amount of stack space is allocated when the query is evaluated. The stack size, the array of bytecodes, and the list of values used in the query are stored in the runtime query representation. In this way, a `query` represents a closure.

Discovery Through the Runtime

When an actor attempts to locate another entity via the `findActors()`, `findStages()`, or `findReplaceableActors()` operations, the specified `query` is sent to all stages in range. If actors are being searched for, an interface is also sent. Network communication is discussed later in this section. In the language, an interface type is specified, however, the compiler will generate a string-based encoding of the interface, using the approach discussed in Section 4.2.3.

Before the query is transmitted, it is encoded. To reduce the size of the transmitted information, all data is encoded once, with each bytecode operation referencing this data. References are 8-bits long, limiting the number of unique values in a query to 256; this limitation has not yet proven restrictive. This optimisation is useful when the same value is compared multiple times, for example, when comparing that a value lies between certain limits.

Once received by a stage, the discovery type is checked. If for actors, the interface encoding is compared against all actors which have been published at this stage. Only actors that have an interface containing channels which match the supplied interface may be queried. An actor may have multiple interfaces, but only one need match. However, if looking for an actor which may be replaced, the specified interface must match all the interfaces of an actor for it to be queried. As the interfaces are represented as strings, the matching is a string comparison. This does not apply to stages as they do not use channels or interfaces.

Next, the boolean expression of the query is executed by a simple stack machine against the values in the query, and either the properties of the local stage, or the properties of each eligible actor depending on the type of discovery.

For actors, the properties are static entities which were explicitly published. A property consists of a key and a value. The remote keys (Section 3.4.2) which are used in the query are place holders for the keys of the properties of the actor currently being examined. If there is a match between the name and type of a remote key and the key of an actor's property, the value of the actor's property is used in place of the remote key in the query. If there is no match, or the types do not match, that particular clause of the expression evaluates to false. Hence, a valid result is obtained if the actor does not have properties which match those used in the query. This is useful as an actor may have similarly named properties with different types. Should the boolean expression which represents the query evaluate to true, a remote reference is constructed for the actor and the channels of its matched interface. This is then repeated for all other eligible actors.

For stages, only its name is a static value. All other values are determined at runtime as they reflect resources which change at runtime, such as the amount of RAM currently available at a stage. Section 3.4.3 describes the currently supported stage properties. In the query, these runtime values are accessed in the same manner as remote keys. However, rather than returning a static value, the runtime will be queried to produce the requested value. If this stage meets the criteria of the query, a remote reference is constructed and returned to the querying actor.

A stage's properties are inherited by the actors who execute within it. Hence, actors may be discovered using attributes of the current runtime environment, as well as their own individual properties. This is useful when looking for existing actors to perform tasks. For example, if looking for an actor to perform processing, it would be useful to eliminate actors on stages who have high CPU usage. Unlike the other static properties, the `#DISTANCE` property is not implemented in the same manner across all platforms. This property is used to represent the physical distance between stages or actors during a query. At present, this is only implemented via Bluetooth, using the Received Signal Strength Indication (RSSI). If not implemented by a stage, the relevant clause will return false. This value can be used in a coarse-grained way to detect locality to other stages, and could be expanded to accommodate the RSSI of wifi, or even use GPS if available. This approach would require a standard definition of distance which could be mapped to the underlying technology. The use of meters would be the most likely choice as they are standardised.

Once all appropriate references have been constructed, they are sent back to the initial querying actor. If there were no references, then no reply is sent. At the initial stage, an array is created and populated with any received references. Note that this array may be empty if no replies are received before the discovery timeout fires, discussed in the following section. This array is then placed on the call stack, and represents the return value of the search. As this is a standard array, it may be used in the same manner as any other array in the language. The VM will appropriately handle the reference counts of any arguments on the stack.

In addition to contacting remote stages, the VM will also query any local published actors and the local stage. In this case, there is no need to encode or transmit any data. The same query process is used as for remote actors. Any eligible actors are added to the list of actor references.

Discovery Through the Ether

The discovery process in Ensemble is based on the idea of *Zero Configuration Networking*, as discussed in Section 2.2.4. This approach was chosen for two reasons. Firstly, given the different equivalence classes of hardware devices, the different networking hardware that they use, and the fact that some hardware platforms are physically mobile, it was not practicable to assume that there would be a central *oracle* (name server/broker) which devices would be able to query and register. Secondly, given that the goal of this work was to explore the use of applications which are reconfigured at runtime, a centralised repository of data may either become outdated quickly, or would require a large amount of network communication to keep the information up-to-date.

To facilitate the discovery of actors and stages described in Section 4.4.1, the runtime uses a range of communication technologies to discover and interact with different physical machines. For platforms using Ethernet or wifi, IP multicast [121] is used to locate the devices concurrently. Each device will then respond via TCP if it has eligible references to communicate. The information required to create a connection is supplied in the initial multicast information. Devices using Bluetooth require a discovery phase to locate all devices in range, and then each device in turn is connected to and communicated with. The Zigbee radio transceiver found on embedded devices uses primitive broadcast and receive actions. These operations can be combined with flood [122] and AODV-based [122] protocols to enable more advanced forms of communication.

Ensemble uses an on-demand approach to discovering other entities at runtime. When an actor performs a `publish`, the list of properties and the visibility of the actor is recorded at the **local** stage. No information is communicated to other physical machines. When one actor attempts to locate another actor or stage, it will broadcast this request through one or more communication media, depending on the radios which are supported on the current device.

It is important to note that the language model does not dictate this approach to discovery, and it would be equivalent to use a dedicated infrastructure. Indeed, for enclosed networks of computer hardware, such as clusters or data centres, it may be appropriate to have a hierarchy of oracles in order to prevent the flood of discovery requests which may occur in the default implementation. This would help as the number of stages increases.

4.4.2 Location Transparency via Channels

It is the responsibility of the runtime to enable channels to be used in a location transparent way. This means that the blocking rendezvous communication model must be replicated in a distributed context to maintain the same semantics as the local case.

The representation which is used for a channel is extended to indicate if it is a local channel, or reference to a channel in a remote stage. In this way, there are few changes required to the non-distributed version of the runtime. Each remote version of a channel keeps a record of a unique runtime ID associated with this channel, the direction of this channel (`in` or `out`), and the stage at which this channel is located. The type of the channel does not need to be specified as there is no way in the language to gain a reference to a channel without knowing its type, hence the compiler will ensure that all interaction with this channel is legal. The rules for using such a channel are described in Section 3.1.3.

Distributed Channel Interactions

To minimise the impact on the runtime, the existing functions which implement the channel operations were extended to accommodate remote channels: no change was made to any function API. Instead, the C structs used to represent channels were modified to indicate if they are local or not. This information is used to determine if a local or remote channel operation is invoked.

In the language, two connected channels will stay connected until explicitly disconnected, or an error occurs. In the runtime, when two remote channels are connected each channel will create a proxy to represent the remote channel and store it locally. As well as information about the other channel, the proxy will store information about the location of the other channel. This includes the network address of the physical machine where the remote channel is located. Unlike the language, the runtime does not use persistent connections between connected channels. Instead, each time an actor wishes to perform a channel operation, a new connection must be made to the remote platform. The format of the network address is used to determine which network technology should be used - i.e. TCP, Bluetooth, Zigbee. Given the unreliable nature of both the hardware platforms and communication technologies used in this work, it would be infeasible to guarantee persistent connections; this is also the conclusion of the Ambient system [123]. Also, the presence of persistent connections in the Erlang runtime between different nodes acts as a limiting factor to the scaling of Erlang applications [124]. Furthermore, migrating actors with open TCP connections would require kernel modifications. Finally, by not being bound to a single communication technology, others could be used if the primary choice is not available: e.g., there is no wifi signal, so try Bluetooth.

Once a connection has been made between two channels, the semantics of `send` and `receive` are the same for both local and remote communication. The only difference is that when the runtime attempts to check the state of a remote channel, it must first create a connection to the remote stage. Additionally, these channel actions may now generate an exception if an error occurs. The possible exceptions are described in Section 3.1.7. If a channel has multiple connections, remote and local channels have an equal likelihood of being chosen.

The logic of `select` is slightly different. The first phase of `select` is to build a list of eligible channels from those specified to be selected from. Should one of these channels be remote, the remote stage will be contacted to get the state of the remote actor. Normally, if this list contained multiple channels which are ready to send, one would be chosen non-deterministically and the data transferred. However, if in the first phase a remote channel is found in a state that can pass information, it will automatically be selected as the channel to be used and its data transferred, without considering other channels. This is required as there is no distributed lock: if a remote channel is found in a usable state in the first phase, by the time it is queried in the second phase, it may no longer be usable, even if other channels are. By this point, the select will have blocked on that remote channel, assuming that it had data. This could lead to deadlock, hence, a remote channel with data will be selected and received from in the first phase. To ensure fairness between local and remote channels in the first phase, the order in which the channels are examined is rotated in a round robin manner between `select` operations. Again, remote and local channels are of equal priority.

Channel operations within a stage are protected by a lock (Section 4.3.4). Hence, it is necessary to regulate the actions of distributed channel operations to avoid distributed deadlock. Table 4.3 describes the interactions of channel operations in a distributed context, and how such deadlock is avoided. Each row indicates the action being performed and how it reacts when it encounters the action specified in the column. ** means that the encounter can not occur.

Remote disconnection is different from other operations as it is a *lazy* operation. When an actor invokes the `disconnect` operation on a channel, any connections to local channels are removed, but only the local references to remote channels are removed; the remote channel is not informed that the connection has been dissolved. Instead, when the remote channel attempts to communicate with this channel, the stage where the channel is expected will reply, indicating that there is no longer a connection. In this way, the disconnection notification is deferred, but still obeys the channel communication rules of the language.

Daemon Actor

When created, a stage will instantiate a *daemon* actor. This actor has no channels. The daemon actor is implemented in C and is responsible for handling incoming discovery, adaptation and communication requests.

As discussed previously, discovery is implemented using zero configuration, rather than using a dedicated infrastructure. The daemon actor is responsible for listening to such messages from the network, processing them, and replying. It is the daemon actor who performs the evaluation of queries against published actors.

For channel operations in general, the daemon actor is responsible for listening and accepting connections. It then either invokes the requested channel operation on the correct channel, or replies to the requester indicating an appropriate error, such as `CHANNEL_NOT_FOUND`. For operations on channels which are found, the daemon uses modified versions of the `send`, `receive`, `select`, and `connect` functions to complete the requested action. The daemon actor is also responsible for accepting and processing incoming actor `spawn` and `migration` requests.

	Send	Receive	Select	Connect	Disconnect
Send	**	Send has priority.	Send has priority.	Connection returns success as send implies connection. Send continues normally.	Disconnect has priority. Send will disregard this channel and remove the connection.
Receive	Receiver waits.	**	**	Connection returns success as receive implies connection. Receive continues normally.	Disconnect has priority. Receive will disregard this channel and remove the connection.
Select	Select waits.	**	**	Connection returns success as select implies connection. Select continues normally.	Disconnect has priority. Select will disregard this channel and remove the connection.
Connect	Connection returns success as send implies connection.	Connect returns success, as the receive means the connection already exists	Connection returns success as select implies connection.	Return success.	Connection fails.
Disconnect	Disconnect returns success	Disconnect returns success	Disconnect returns success	Disconnect returns success	Disconnect returns success

Table 4.3: Result of Simultaneous Distributed Channel Operations

As the daemon actor is used to demultiplex incoming messages to the relevant actor, there may only be a single stage per physical machine because the daemon will use the network address (IP/Bluetooth) of the machine that it is at to be contacted. Hence, to support multiple stages per machine some mechanism to demultiplex incoming messages to stages is required. Then the daemon within each stage can deliver messages to the correct actors. This is a similar problem faced by MPI [57]. In MPI, this is solved by the inclusion of a daemon process where each new MPI process must register with this daemon at creation. The daemon will then forward relevant messages to the appropriate process. In this way, a single network address can be shared, and multiple stages may exist per physical machine. A similar scheme is envisaged for this work.

Marshalling and Demarshalling

In order for data to be sent between remote actors it must be translated to a stream of bytes and then reconstructed at the remote end. The runtime will automatically marshal and demarshal any data type in the language, language defined or otherwise, without intervention from the user. There are many different approaches to this, including a number of libraries for C[2], Python[3], and Java[4]. Rather than use an existing library, which may not even fit on some of the targeted hardware platforms, custom data marshalling is used. This also benefits from being tightly integrated with the VM.

Data marshalling is based on the existing reference counting system. When an object is to be garbage collected, its references are visited. This will continue until a language defined object is encountered. For each language type, there is a pre-defined destructor function. For marshalling, an equivalent set of functions are defined which encode the type to a stream of bytes. Hence the same graph traversal is used. This graph traversal is done twice: once to calculate the amount of space required for the marshalled data, and once to encode the data.

At the receiving end of a remote channel, there is no reference graph to follow. Instead, the string encoding of the data type the channel conveys (Section 4.2.3) is used to reconstruct the object(s). This has the advantage that neither the type nor associated classes need to be transmitted, as the compiler gives two guarantees. Firstly, two connected channels will convey data of the same type, meaning that data will always be decoded correctly. Secondly, as the data type conveyed by the channels must have been defined prior to the declaration of the channels, the class files representing the transmitted data type will be available at both the sending and receiving stages. Hence, class files need not be transmitted with the data itself. This is a strong advantage of using typed channels, rather than sending all data types

[2]https://github.com/protobuf-c/protobuf-c - Accessed March 2015
[3]https://docs.python.org/2/library/pickle.html - Accessed April 2015
[4]http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html - Accessed May 2015

to a single actor's mailbox directly, as in many actor languages.

This said, one complication is a channel which convey data of the `any` type. In this simple case, the receiving channel will only need to know that the data is of the `any` type, hence of type `Object`. However, should the application ever wish to access the inner data type, it must be reconstructible. Consequently, when data of type `any` is sent, both the data and an encoding of its type must be sent. The inner data type will then be reconstructed using the encoded type information, rather than the type of the channel. Using this information, the data will be reconstructed if an appropriate class can be found, otherwise, the data will remain in an encoded state. The resulting data will be wrapped in an `Object` to create an `any` type. This data type can be used as normal, including being used with the `project` statement.

The project statement uses the `instanceof` instruction to determine if two objects are of the same type. From a linguistic point, this guarantees that the application either has a definition of the `any`'s inner type, and can be used safely, or the inner type cannot be accessed. For the VM, this means that there is a class file for the inner data type. If no class file is found, the instance of the `any` (`Object`) type can still be assigned to or sent to other actors, but the inner type is inaccessible.

4.4.3 Actor Adaptation

Given the encapsulation of actors, they represent the perfect unit to enable modular program composition, but also the perfect unit to enable runtime adaptation. Ensemble enables this to be expressed natively as a part of the language (Section 3.4), hence the VM must support this. This adaptation is split into *spawn*, *migrate*, and *replace*.

Spawn

The purpose of `spawn` is to create a new actor. Unlike the `new` operator, spawn is responsible for creating a new actor at a specific stage. This stage can either be the same as the current stage, but more likely a different stage on a different physical machine. In either case, the process is the same from a linguistic perspective, however differs within the runtime. Also, unlike a `new` operation, a spawned actor may be created with or without a reference to it. This either replicates the `new` operation in a potentially distributed context, or offers a *fire and forget* approach to actor creation. The latter can be useful for a factory actor which is asked to create an actor, but does not need to reference it.

The process of spawning an actor at a stage consists of transferring the class files of the actor, any initial state, and the class files upon which the actor depends to a specified stage. Once all state has reached the remote stage, a new instance of the actor is created.

Initially, a connection is made with the specified stage using the information stored within the stage object which is passed to the spawn statement. If a connection cannot be established, an exception is generated at the spawn call site, and the spawn is aborted. If the connection is successful, the classes which the actor is dependent upon, as described in the actor's class file (Section 4.2.3), are encoded for transmission along with the class for the actor.

To ensure that all transitive dependencies are met, the dependencies for all types must be found and encoded for transmission, including those not directly referenced by the actor. This is done recursively at runtime. As described below, Ensemble does not use on-demand loading of classes like Java, instead requiring that all class files be present before an actor is executed at a stage.

In addition to the class files, if an actor to be spawned is passed any values or variables via its constructor, these will be marshalled using the approach described in Section 4.4.2. This data will then be added to the encoding along with which particular constructor should be invoked.

Assuming successful transmission, the remote stage will first decode and allocate space for each transmitted class which does not already exist at the stage. Each stage maintains a list of all loaded classes. The remote stage will then create a temporary stack. Any values or variables which have been supplied for the actor's constructor are pushed to the stack. The constructor will then be invoked, and the actor created in a paused state. No type-checking need be performed at this point, as the Ensemble compiler has already guaranteed that any values are legal for the specified constructor. After the actor is created, the temporary stack and its contents are garbage collected. The remote stage will then communicate to the actor that invoked the spawn that the new actor was created successfully. Both local and remote actors will then continue execution. Otherwise, the remote stage will indicate why the actor was not spawned, perform the necessary garbage collection, and a relevant exception will be generated at the spawn call site. The set of potential exceptions are the same as when invoking a `new` operation, with the addition of the `StageNotReachableException`.

The above describes the process for actors which are spawned *without* any reference. For actors which require references, once the new actor has been created in a paused state, the remote stage will create a reference for it and its channels. This is the same type of reference as is returned for actors found using the discovery mechanism. This reference is then transmitted to the invoking actor together with the confirmation that the new actor was spawned successfully, and both actors continue with their execution. Otherwise a relevant exception is generated as before. Should the specified stage be the local stage, the runtime will treat this as a special case and create the actor locally in a similar manner to a `new` operation.

An obvious optimisation would be to add an extra step before the transmission of the data which conveys the classes which are about to be transmitted. Here, the remote stage could indicate which classes are already present, thus potentially reducing the number of classes which need be sent. Unlike the JVM, which uses on demand loading of classes, the Ensemble runtime ensures that all required classes are present at a stage before an actor is instantiated. Although this approach may transfer classes which are not used, it ultimately simplifies the system, and guarantees that all dependencies will be met at the completion of the spawn. This is the same approach as in Emerald (Section 2.1). Given the unreliable runtime environment being explored in this work, it may not be possible to load a class from across the network at a later date, as the physical device may no longer be accessible. Also, it was decided that exposing class loading errors would unnecessarily complicate the language.

Migrate

Migration of an actor is where an executing actor will pause its execution, relocate to a specified stage, and then continue execution. As described in Section 3.4.4, this is supported natively in the language, and enables fine-grained adaptation of Ensemble applications.

As in a spawn, the actor's class files must be transmitted, but also the state that it possesses and the stack upon which it is currently executing. It is important to note that this state includes all channels and their connections. This is an example of strong migration.

To migrate an actor, the classes upon which the actor depends are encoded in the same manner as for spawn. Next, the state of the actor, including its channels, are encoded using the marshalling mechanism discussed in Section 4.4.2. This includes any messages which are in the buffer of an *in* channel. After the classes and the state, the actor's stack is serialised. The stack represents the current execution state of the actor and consists of the call frames of any invoked procedures plus a frame for the behaviour clause.

The data to migrate an actor is encoded as a data pool and a stack. The data pool contains the marshalled data items, and the stack consists of the stack frames, with offsets into the data pool for any object references. This saves space for objects which are referenced multiple times.

Once the transmission is received, the data objects, including the actor, are reconstructed. Also, the reference counts for each object must be restored. If the counts are not reconstructed correctly, this will cause memory errors as the actor executes and attempts to release memory to the heap naturally. At this point, the stack is rebuilt, with all values and variables popped onto the stack in the correct places. The interpreter is then pointed to the bytecode after the bytecode which invoked the migration, and the actor is placed in a paused state.

Once the initial stage has received confirmation that the actor has migrated successfully, the final action is to delete the local actor, return any resources which it used, and tell the migrated actor to continue execution.

Note that migration must maintain any channel connections between both the actor being migrated and the actor connected by the channel. This is the responsibility of the migration operation after the actor has been reconstituted at the new stage. While the actor is being migrated, it is placed into the `MIGRATION` state. This is done to prevent channel operations reaching inconsistent states: for send attempts, the actor will not be in a state which is eligible to receive data, hence the sending actor will either try another connection, or block. For connection attempts, the actor will appear as being not found, and generate an exception at the connecting actor (Section 3.4.5). In this case, the connecting actor must try a different channel or perform another discovery to locate it at its new stage. Once the actor has been reconstructed at the new stage, it is placed back into the `RUNNING` state, and must then recreate any connections which existed before the migration. This uses the existing connect mechanism. Should the actor now be co-located to any previously remote channels, both ends of the connection will be given local channel representations to increase performance.

If an actor requests to migrate to the local stage, migration does nothing and execution continues locally.

If there are any exceptional conditions, such as connection failure or lack of RAM on the new stage, the migration will be aborted and an exception will be raised at the migration call site. Any allocated resources at the remote stage will be released.

Replace

Section 3.4.4 describes the process and requirements on locating and replacing actors. Although this is implemented in the language, it is not completely implemented in the runtime. Locating replaceable actors is supported, but the actual replacement is not.

Once appropriate actors have been found, replacement is envisaged as an advanced form of spawn. As well as relocating the relevant class files to the new stage and creating the actor, the new actor must also assume the channel connections of the existing actor. As described in the language, no consideration need be made for the state of the existing actor.

The one potential flaw with this approach is regarding channels which have been created at runtime, rather than declared in the interface. As these are not visible during discovery, even with `findReplaceble()`, there is the potential to cause unforeseeable application level logic errors. There are two points to consider here:

Firstly, any channel which is sent across another channel is not directly usable by the sending actor. Either a duplicate of the channel is sent, in which case two distinct channels are created, or the channel was movable and cannot be used until a new assignment is made. As the sent channel is unusable by the sending actor in a meaningful way, replacement will have no effect. Secondly, if two opposite channels are created at runtime from the heap, connected together, and one is sent to another actor, a problem would exist. In this case, the channel which is not sent can be used to communicate in a meaningful way with the other channel which has been sent. As such channels are not present in an actor's interface, there is no way to know of their existence using the current approach. Should such an actor be replaced, the interaction with other actors could become broken. This can be avoided by always declaring the channels to be used within the interface, and is a coding idiom of the language. However, this represents a limitation of this approach.

As well as knowing which actors can be replaced, it is important to know *when* an actor can be replaced, such that the logic of replacement can be reasoned about. When an actor invokes the `stop` statement, it will exit the next time it completes all code in its behaviour clause, at which point the actor's resources are returned. By having the execution finish at this point, it simplifies the reasoning of stopping the execution of this actor, and its interaction with others. The same logic will be used for replacement. When an actor is to be replaced, it will exit after having completed executing all code in its behaviour clause. At this point, a new actor with the same channels and connections would be created before the original actor is destroyed. Unlike stop, the replacement of an actor may be performed by another actor, as well as by itself.

4.4.4 Stage Adaptation

Currently, stages exist in a single physical location and cannot be migrated. However, Section 3.4.6 described the potential advantages to moving a stage. To this end, stage migration can be considered as the migration of its constituent actors and its name from one physical location to another. As the mechanisms exist to migrate actors, the process of migrating a stage is mostly implemented and would only require some functional abstraction to carry out the procedure. Given time constraints, this was beyond the scope of this project, however, would be useful in future work to enable exploration of the migration of computation at both the fine grained actor level, as well as the coarse grained stage or application level.

4.5 OpenCL Integration

As described in Section 3.3, Ensemble uses actors to abstract accelerator-based programming of parallel hardware platforms, such as GPUs and multicore CPUs. Specifically, Ensemble actors are used to represent OpenCL kernels (Section 3.3.1), with the explicit data movement between the host and parallel hardware device (accelerator) being represented by channel communication. The remaining boilerplate code is automatically handled by the runtime. In this way, the actor model of computation is used to simplify kernel-based concurrency.

In order to add OpenCL support into the Ensemble runtime, a number of modifications were made. Firstly, support for OpenCL is optional, and conditionally compiled into the runtime. This was necessary to ensure that RAM and ROM on resource-constrained platforms without OpenCL support were not unnecessarily consumed. In this way, the same source tree is still available for multiple platforms, reducing maintenance and update effort.

Secondly, during the initialisation of the runtime, a single matrix is created to hold references to the different platforms and devices available in this system. This is done to ensure that there is only a single `command_queue` per device, rather than each kernel actor creating a new one. This was necessary as race-conditions were observed with multiple `command_queues` per device when reading data. The information passed in the declaration (Figure 3.13, line 21) of an OpenCL actor is used to index into this matrix at runtime to determine the appropriate `context` and `command_queue`. If no information is given in the declaration, default values are used.

Thirdly, OpenCL wrapper functions were created and made available to the interpreter to simplify and abstract the interaction between the VM and the OpenCL API.

4.5.1 Interpreter

Within the interpreter, all OpenCL operations are implemented in C for performance. Each operation described in Section 3.3.1 is implemented as a custom native operation in the VM which is invoked by the `invokenative` bytecode. Also, each OpenCL actor is given an `OpenCLEnvironment` variable. This is a runtime structure only visible within the interpreter that is used to store metadata about the platform, device, and device type, as well as the relevant `command_queue` and `context` for a given OpenCL actor. This structure is populated when the actor is created using the information contained in the previously described runtime matrix.

4.5.2 Lazy Evaluation

In OpenCL, a common idiom is to leave data on a device for as long as possible, thus reducing the time spent copying data between the device and host, and ultimately the application's execution time; data movement is often the largest performance bottleneck. In actor-based languages there is no shared state, hence when messages are sent between actors, a duplicate is created and sent. This ensures no shared state between the actors, and that each actor has a unique copy of the data. While this is correct, it costs time, requires greater memory consumption, and precludes keeping data on the device.

To prevent such duplication, Ensemble supports marking non-primitive types as movable (`mov`), as discussed in Section 3.2. This approach has been applied to the OpenCL kernels in Ensemble. Marking the `in` channel to the kernel actor used for data as moveable (`mov`) has two effects. Firstly, once any non-primitive data is copied to the device it is marked as no longer being on the host, and copies of relevant OpenCL data structures are associated with the runtime representation of the data type. Secondly, the compiler will not generate the code to read this data back from the device. Thus, when the data is sent onwards it will hold a reference to the data which is still on the device.

At this point there are two possibilities for the data that is sent onwards. The first option is that the data successfully arrives at another OpenCL actor executing on the same device without being accessed by the host. In this case, the pointer to the device data is set as the appropriate kernel argument, and the kernel is dispatched. Here, the data was kept on the device at all times. The second option is that the data is either accessed directly by host code, or the data is sent to an OpenCL actor associated with a different context. In both cases, the runtime reads the data back from the device and returns the device memory. As Ensemble uses automatic garbage collection, should the host reference count ever reach zero, both host and device memory will be returned. In this way, the choice to use `mov` gives the user control over memory usage.

One benefit of lazy evaluation is that the runtime can automatically determine, using the `OpenCLEnviroment`, if the incoming data needs to be moved to the current context, and then do so if required. Currently, OpenCL manages data movement between devices in a single context, but not in different contexts.

4.5.3 Multiple Implementations

Currently, the ensemble compiler will only generate a C representation of the kernel specified in an actor. This kernel is stored as a string in the actor's class file. As a kernel actor is represented as normal class file, it is possible to `spawn` kernel actors to stages which support OpenCL. A kernel actor may not be migrated as OpenCL does not enable access to the intermediate results of a computation, hence the state could not be captured and then rebuilt on a different hardware device.

One avenue of future work would be to have the Ensemble compiler generate equivalent kernel logic which could be interpreted as a normal actor by the VM. Thus, when a kernel actor is spawned at a stage which does not support OpenCL, it could still execute. This non-OpenCL implementation may be sequential, but could also be threaded in order to take advantage of multiple CPU cores at the current stage without the need for OpenCL. This would also enable research into runtime load balancing between different implementation of the same kernel. Here the runtime could find the most optimal implementation for a given set of operating constraints.

4.6 Heavily Resource-Constrained Platforms

The previous discussion in this chapter has described how the actor-based computation of Ensemble applications is executed on a number of different hardware platforms. One hardware platform class which is often ignored and unsupported by most other programming models, actor or otherwise, are small, resource-constrained battery-powered computers. In order to show that the actor programming model is appropriate in the smallest of execution environments, it was necessary to implement a version of the VM on a highly resource-constrained hardware platform (mote).

This was done on the Tmote Sky platform [111] which has 10KB RAM, 48KB ROM, and uses the MSP430 microprocessor (16-bit 8MHz). Given the small amount of RAM and ROM available on this device, it was not possible to support all of the features of Ensemble. Instead, only the base version of the language without adaptation, accelerator-based concurrency, or location transparent channels is supported. Also, the compilation process and execution model had to be modified. Despite these modification, realistic and useful actor-based applications may be run, see Section 5.1.1.

The following describes the modifications which were made to the Ensemble VM in order to accommodate these highly constrained platforms.

4.6.1 Compilation and Linking

To execute on embedded hardware, the Ensemble VM was modified to the use the *split-VM model*, similar to the VMs discussed in Section 2.2.2. A traditional JVM uses *lazy loading* of class files. Classes are compiled independently, and all references to other classes are symbolic. The JVM loads a class file when it is first referenced, and resolves all symbolic references before continuing execution. However, this is a demanding process, and class files are often larger than the whole main memory of a sensor node.

The essence of the split-VM approach is to resolve all references offline, on a more powerful machine, and link the class files into a single file which the VM can execute. This places less demand on the embedded hardware. The linked file can be smaller than the original class files by an order of magnitude or more. As the linker has resolved all dependencies, the constant pool has been removed from Ensemble class files for this platform. Table 4.4 shows the savings gained by the new linking process. The removal of the constant pool provides significant savings. Furthermore, in addition to the existing techniques used in

Data	Standard library (26 class files)	Programs (37 class files)
Constant pool	68.1%	64.0%
Class metadata used	0.3%	0.2%
Class metadata unused	5.9%	4.1%
Field metadata used	0.3%	0.2%
Field metadata unused	10.9%	2.5%
Method metadata used	1.9%	2.1%
Bytecode	2.4%	13.0%
Method metadata unused	10.3%	13.9%
Method total	14.6%	29.1%
Total used	4.8%	15.4%
Total unused	95.2%	84.6%

Table 4.4: Average percentage composition of class files. 'Used' and 'unused' indicate whether information is present in the modified class file.

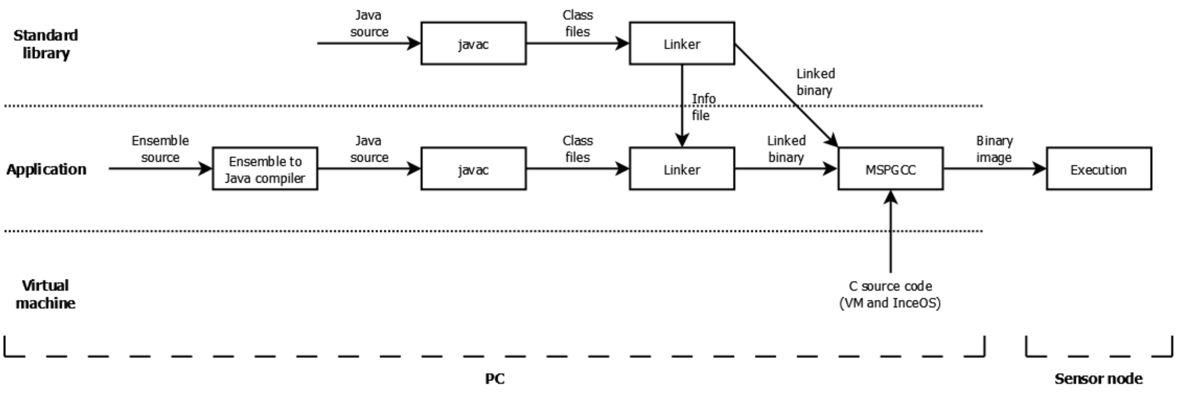


Figure 4.3: Steps to compile and execute an Ensemble application. The 'split-VM' architecture is shown in the separation between PC and sensor node.

the linker (Section 4.2.2), the dependency list is no longer part of the class file. Finally, even though actor and channel adaptation is not supported, string-based type encodings are required at runtime to decode `any` types for inter-stage communication, but in a limited fashion (Section 4.6.3). Channels may still be sent between actors, but only locally.

The split VM system used by the Ensemble VM has five steps, shown in the *application* section of Figure 4.3.

1. Ensemble code is compiled to Java source code. Each actor and type is a Java class.

2. The Java source is compiled to Java bytecode using `javac`.

3. The linker is run twice; first to link the standard library, and second to link the bytecode program against the standard library.

4. The VM is compiled using MSPGCC. The output of the linker is statically compiled into the C program at this stage, and stored in program memory

Table 4.5: Mapping from Ensemble to Java to MSPGCC data types.

Ensemble type	Java type	Size (slots)	MSPGCC type	Size (bits)
integer, unsigned, boolean	int	1	int	16
byte	byte	1	char	8
real	double	2	float	32
long	long	2	long	32
reference	reference	1	void*	16

5. The resulting binary is installed on a mote or run in simulation.

The linker produces a second output: a symbolic information file which allows other applications to link against the corresponding binary. This is used to separate the standard library from applications. The standard library contains classes which applications rely on, such as `Object` and `Integer`, native method declarations, and wrapper classes for system actors.

Conditional Compilation

InceOS provides many standard library functions and actors, only some of which may be used by a given application. The VM cannot simply ignore functionality which is not used by Ensemble applications, and must reference everything, which uses most of the Tmote Sky's 48 kB program memory and leaves little space for bytecode.

This is overcome using conditional compilation in the linker and the VM. Only the core of the system is enabled by default, and the user enables any additional actors required. These actor provide access to sensors, physical storage, and radio communication. Because the standard Java toolchain does not support conditional compilation, a custom *@ifdef* annotation is used by the linker to mimic the behaviour of the C preprocessor directive. The VM build system ensures that a matching set of system actors is compiled. Any interpreted program which uses only these actors can then be run without having to modify the VM image or the standard library installed on a mote. Currently, this process is manual, however, it could be automated by using the compiler to analyse applications to determine which system actors it depends upon, and include these in the compilation. As discovery is not supported on this platform, system actors are visible in all scopes.

4.6.2 Memory Model

As the Tmote Sky is a 16-bit architecture, the Ensemble VM was modified from 32-bit to 16-bit to match the native word size of the MSP430 microprocessor. Table 4.5 describes the mapping between Ensemble, Java, and MSP430 types in terms of the number of VM slots consumed and the number of bits required to represent them. Also, as the VM now has a smaller word size, a stack frame only consumes 16 bytes of RAM, with each additional slot or variable requiring two bytes. No changes were required to the bytecode instructions as they are defined in terms of slots, and not words.

As with the other platforms, objects and stack frames are allocated from the heap as required, and the reference counted garbage collection provided by InceOS is used.

4.6.3 Explicit Communication

Space limitations prevented the use of channels to abstract inter-stage communication. Instead, a well defined system actor is used to enable explicit communications with other stages: the *radio* actor. The radio actor is an actor which is written in C, and executed by a thread. Like the daemon actor, it is created when the runtime boots.

Before interacting with the radio actor, a user actor must first construct a `radio_packet`. This is a language defined structure which consists of an `integer` address field, and an `any` payload field. A user actor must create such a structure and populate it with the network address of the physical node to be sent to, and the data to be sent. For wireless sensor networks, sensor nodes are commonly numbered using integers from 0 to the number of nodes in the network.

Once constructed, a `radio_packet` is sent to the radio actor via a channel. The radio actor currently provides the *broadcast*, *unicast*, and *received* channels. The broadcast and unicast channels accept `radio_packet` types. Any data sent on the broadcast channel will be broadcast to all motes in range. Any data sent on the unicast channel will be transmitted to the mote addressed in the `radio_packet` if it is in range. User actors may use the receive channel to listen for either type of communication. In this way actors and channels can also be used to provide explicit inter-stage communication. If necessary, a similar approach could be used with other platforms.

Communication does not yet support reliable or multi-hop communication, however it may be implemented in user space. Also, the number of channels which the radio actor presents could be expanded to included implementations of reliable communication, or multi-hop routing protocols to deliver packets to arbitrary points in the network. The radio actor itself is *well defined*, meaning that it may be referenced by any Ensemble actor without needing to first locate it.

Note that as the information in a `radio_packet` is an `any` type, it must first be projected in order to access the inner-data type. The string based encoding is used in a similar way to the main Ensemble VM.

4.7 Security

Security considerations were beyond the scope of this work. However, given the ability to spawn/migrate/replace actors at runtime, as well as the concept of having actors from different applications executing within the same stages, there is the possibility for *foul play*. Hence, there is a need to have some consideration of security.

As the language is designed to be simple and usable by non-computer scientists, there was a strong motivation not to complicate the language with security protocols: for example, it may have been possible to annotate a channel to use a specific security protocol. Instead, it is preferable to use the runtime to provide a safe operating environment, within which applications can operate.

4.7.1 Communication

As Ensemble uses existing networking technologies, there are existing approaches to secure IP [125], Bluetooth[5], and Zigbee [126]. Instead, protection would be required at the stage level. Both Erlang and MPI's default security model is to use a shared password between nodes which is supplied when a node is instantiated. Nodes which share a similar password may communicate, where those who do not may not communicate. This technique can also be used for stages in Ensemble. Should two stages not share the same password not only will actors and stages not be discovered, but if a stage/actor is found by an eligible stage, and then shared with an ineligible stage, the ineligible stage would not be allowed to communication with the eligible stage.

4.7.2 Execution

There are a number of issues with regard to having actors which may be reconfigured at runtime:

- Although the system enables actors from different applications to work together on the same stage, they are all isolated in separate memory spaces by the VM. Hence actors from different applications can safely execute in a stage.
- Java does not enable access to the state of running threads due to security concerns, hence migration may not be implemented natively. Although Ensemble provides migration, it is a native operation. This state is accessed by the runtime, and will never be accessible by user code.

[5]https://www.bluetooth.org/en-us/specification/adopted-specifications - Accessed February 2015

- As actors are capable of spawning other actors, *actor (fork) bombing* is possible. This can be mitigated by limiting the number of actors within a given stage, or the rate at which actors are spawned/created.

- It is possible to create an actor in a stage which is performing useless intense computation with the purpose of preventing other actors from doing useful work. All Ensemble platforms currently support pre-emptive multi-threaded execution, hence even if an actor is not manually blocking, the runtime will ensure fair execution for all actors.

4.8 Summary

In order to support the actor-based programming model in the Ensemble language across heterogeneous hardware platforms, applications are first compiled to Java source code, and then to bytecodes using the existing Java compilation process. The resulting class files are then processed by a custom linker to reduce their size, and remove unused Java features. These bytecodes are then executed on a custom-built virtual machine, which has been ported to a number of different hardware platforms from different equivalence classes of system scale.

In order to support the distribution and adaptability expressed in the language, the runtime was expanded to enable discovery of actors and stages at runtime in a decentralised way, location transparent inter-stage channel communication, as well as the creation and relocation of actors between different physical hardware platforms. This also included reporting failure in a meaningful way.

Additionally, the runtime was extended to include support for the OpenCL framework, where actors represent kernels. As well as automating much of the boilerplate code required to initialise OpenCL devices, the channel communication mechanism was modified to abstract the movement of data to and from an accelerator. By using the existing compiletime analysis for movability, lazy evaluation was employed to reduce the amount of data movement.

Finally, to explore the application of actor-based programming at the smallest scale of computing, the VM was ported to a wireless sensor device. This required the use of the 'split-vm' approach, whereby an application is linked on a powerful machine, leaving a statically linked binary which is loaded onto the sensor mote. Space constraints prevent actor reconfiguration, but do allow a suitable environment for sensor network applications.

Chapter 5

The Actor as the Unit of Abstraction

The previous two chapters have described a programming language based on the actor model of computation designed to simplify programming of concurrent, distributed, and adaptive applications across different equivalence classes of hardware platform, as well as the design and implementation of a runtime to facilitate the model expressed in the language.

The purpose of this chapter is to apply the actor model, as expressed in the language and runtime, to the areas of concurrent, distributed, and adaptive computation across different hardware platforms in order to show that the actor is the correct unit of abstraction when programming embedded, highly parallel, and heterogeneous systems either in isolation or in concert.

Before the discussion, it is worth noting that any application written using actors can also be written in any other Turing complete language. The power of actors is in the structuring of the application. By forcing a developer to express their application in terms of autonomous loci of computation that interact with explicit communication, the application is loosely coupled, modular, and naturally suited to concurrency, distribution, and adaptation.

This Chapter is split into five sections. Section 5.1 discusses the actor as an appropriate unit of abstraction for programming embedded systems, specifically WSNs. There is a comparison between this work and TinyOS, the de-facto standard in the field, in terms of quantitative linguistic complexity and runtime performance across a range of applications. There is also an analysis of the runtime impact of movable types. In Section 5.2 the actor model is shown to be an appropriate abstraction level for programming parallel hardware platforms, by simplifying the use of the OpenCL framework. Again, there is a quantitative linguistic comparison and performance evaluation on a number of applications. The use of the actor as the unit of adaptive programming across heterogeneous hardware platforms is discussed in Section 5.3. Its use is motivated by a number of examples where adaptation is required, as well as a performance analysis showing that the advantages of adaptation is greater than the cost. Finally, Section 5.4 gives a summary of the points made in this chapter.

5.1 The Actor as the Abstraction for Embedded Programming

Resource-constrained embedded systems are being increasingly embedded in our surrounding environments, to both collect data from and effect changes in such environments. Even watches are now powerful enough to execute non-trivial computation, and are advancing the definition of embedded hardware[1]. Wireless sensor networks are one category of resource-constrained embedded system which exhibits all of the traits found in the domain; limited battery power, RAM, ROM, and processing power. In this work, WSNs have been used as a case study for the application of actor-based programming to embedded systems, with the results being generally applicable to systems with similar constraints.

Wireless sensor networks enable a wide variety of activities to be performed autonomously, and are currently being used in many diverse areas including measurements of mountain permafrost [127] and grapevines [128]. Such networks take highly constrained hardware devices (motes) and connect them via short-range radios to form useful monitoring tools, protection systems, and research systems. In programming such devices, a number of unconventional programming models have evolved. For example, the TinyOS [30] and Contiki [129] embedded operating systems use an event-driven programming model, although realised through different abstractions.

Event-driven systems respond to events. These events can be generated by the hardware, for example by interrupts, or by software. An event triggers an associated event handler, which handles the event and results in some computation being initiated that may, in turn, generate further events. In event-driven systems, there is no single locus of execution; rather, there are a number of them each triggered by an event. Such systems have become popular for embedded systems since they do not require the same memory and processing overheads as threads (e.g., for stacks and context switching), yet provide a concurrent computational model. Another advantage is that concurrency control is simplified since, in many (single-CPU) systems, multiple event handlers do not run simultaneously [30].

Widely-used operating systems for wireless sensor networks impose unusual programming models to compensate for the limited resources available on embedded hardware platforms. For example, TinyOS [30] uses the nesC language, with an event-driven 'split-phase' programming model. In nesC, all operations are non-blocking, and programs use many callbacks which can make the flow of control difficult to follow [22]. Equally, Contiki [129] programs are written in C, but use macros and continuations to simulate a traditional threaded environment on top of an event-driven core. Other approaches, such as MagnetOS [130] and

[1]http://www.apple.com/pr/library/2014/09/09Apple-Unveils-Apple-Watch-Apples-Most-Personal-Device-Ever.html - Accessed May 2015

SwissQM [131] also support unusual execution models. MagnetOS treats the whole network as a single VM, and SwissQM treats it as a database. Programmers using these systems must have extensive knowledge of low-level and embedded programming. Domain experts wishing to use WSNs in their own fields – often described as the intended users of these systems – are unlikely to have this knowledge.

These models are non-intuitive to programmers due to the introduction of unnecessary non-intrinsic complexity. In particular, the introduction of the TinyOS split-phase execution model is a barrier to understanding, writing and reasoning about WSN programs. The same argument could also be levelled at programming with TinyOS threads [132] or Contiki protothreads [133], but is not included for brevity.

This work hypothesises that the use the actor-based programming model simplifies development for resource-constrained embedded systems, removing much of the complexity of even-based programming, while still affording the developer the power to create complex applications. Also, the use of a VM specifically designed to execute such applications does not prevent their use in resource-constrained environments. This is shown by implementing such applications in Ensemble, and executing them both natively and by the Ensemble VM on the Tmote Sky.

5.1.1 Applications

In order to evaluate the complexity and performance of an actor-based approach to WSN applications, the following examples where chosen:

- *BlinkA* is a simple application that periodically blinks the three different LED's of the Tmote Sky at different rates. *BlinkB* performs the same operation using the TinyOS thread library (*TOSThreads* [132]).
- *TestSineSensor* periodically samples a sensor, after which it forwards the obtained value over the serial link. Under TinyOS, it is implemented using TOSThreads.
- *RadioStress* uses three threads to send messages to another mote where three threads are listening for messages from their counterparts. Under TinyOS, it is implemented using TOSThreads.
- *RadioCountToLeds* involves two motes, one maintains a counter which is transmitted over the radio to the other mote which displays the lower three bits of the transmitted value on its LED's.
- *RadioSenseToLeds* is a similar application, except that it collects and sends sensor data as opposed to a software counter.

- *Sense* is similar to RadioSenseToLeds, but it only uses one mote and does not send sensor values over the radio.
- *TestRoundRobinArbiter* is an example of an access control mechanism where three resource users request access from a central controller, which grants access to each in turn.
- *Fourier* performs a Fourier transform on an array of 40 integers repeatedly.
- *Grid* is based on the notion of using ad-hoc grids in sensor networks to mitigate the power consumed by excessive radio transmission of data [134]. This application consists of two types of node: leaders, who make requests, and slaves, who service requests. Initially the leader broadcasts a request asking for any free slave. Once a slave replies to this request, the leader collects enough sensor data to fill an array of size 10 and transmits it to the slave that acknowledged it. The slave performs a Fourier transform on the received data, calculates the maximum value and returns this to the leader.
- *DataLogger* is a prototype application to monitor the effects of fluid flow on riverbed sediment. This require continuous measurement of four different sensors at 50Hz, logging this data to storage, and radio communication between the sensor collecting the data and a user controlled base-station for data collection and reporting. This program uses many of the optional actors provided by the standard library, including the acceleration and tilt sensors, the radio, the storage module, timers, and runtime error checking.

The Grid, Fourier, and DataLogger applications were not provided by TinyOS, but all other were. The combination of these applications cover intense computation, radio transmission, interactions between actors on a single node and combinations thereof, and are representative of the typical actions of applications on this type of device. In particular, Datalogger represents a typical embedded application, where data is sampled, stored, and reported over the radio.

5.1.2 Experimental Setup

All experiments were performed on the Tmote Sky hardware with fully charged batteries at the start of each experiment. Performance results were confirmed on the Cooja simulator [135].

	Lines of Code		Actors/Components		Wiring Statements		Interfaces	
Application	Ensemble	nesC	Ensemble	nesC	Ensemble	nesC	Ensemble	nesC
BlinkA	28	40	2	6	3	5	1	5
BlinkB	30	54	4	7	1	6	1	6
TestSineSensor	13	45	2	7	2	8	1	7
RadioStress	50	94	5	13	9	12	1	12
RadioCountToLeds	55	103	5	7	4	7	2	7
RadioSenseToLeds	57	101	6	8	6	8	2	8
Sense	29	43	3	5	3	4	1	4
RoundRobinArbiter	49	180	5	11	10	15	2	24
Fourier	19	30	1	2	0	1	1	1
Grid	97	177	5	8	7	8	1	8
DataLogger	567	2552	5	22	20	37	5	68

Table 5.1: Comparison of Ensemble and nesC code.

5.1.3 Code Complexity

Both Ensemble and nesC applications are composed of components/actors, interfaces, and wiring statements. Accordingly, these features were used as metrics to judge code complexity. Table 5.1 shows the results. The table shows the application, number of lines of code used, number of components/actors either written or referenced, number of wiring/connect statements, and the number of interfaces used. Obviously each of these features could be manipulated -e.g., every Ensemble application could be written in a single actor. To prevent this, each Ensemble application uses an actor representing each activity of the program and the system actors.

The table shows that the applications can be written in Ensemble with fewer elements from each category, excluding the Fourier application where both languages require one interface. Although it does not necessarily follow that fewer is better, the previous discussion of the simpler composition of Ensemble and these results show that it is possible to write functionally-equivalent programs in Ensemble which are simpler.

5.1.4 Memory Usage

The initial implementation of Ensemble directly generated C code which was compiled with InceOS to generate a binary. This binary was then uploaded to a sensor mote. The RAM and ROM requirements of using this method are described in Section C.1.1. As the purpose of porting the Ensemble VM to the Tmote Sky was to provide a platform to enable runtime adaptation, the following results are for the Ensemble VM.

The static memory usage of Ensemble applications has been examined in Section 4.6.1. Ensemble applications also allocate memory dynamically from the heap. Because the garbage collector uses reference counting, memory is freed as soon as it is no longer referenced.

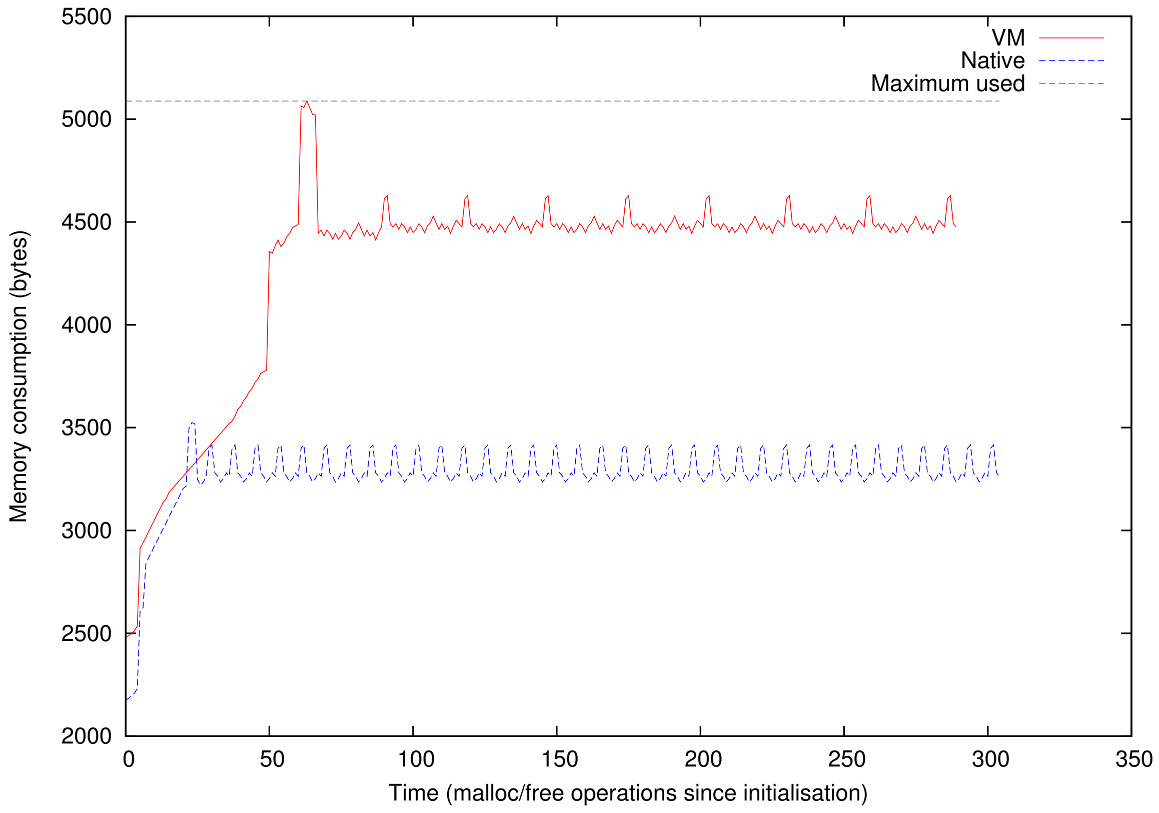


Figure 5.1: Dynamic memory usage of the *RadioSenseToLeds* application.

RadioSenseToLeds

In Figure 5.1, the 'VM' plot shows the dynamic memory usage of the *RadioSenseToLeds* application over time, running on the Ensemble VM. Time is measured in the number of allocations and deallocations since the end of the OS initialisation sequence.

Memory tracking starts at the beginning of the `boot` clause, after system initialisation. Most of the memory allocated before this is used for the C call stacks of the system actors and the stage. The large increases at around times 50 and 60 are the creation of the sender and receiver actors (again, most of the memory is used for the C stacks). The large drop at time 70 is the termination of the actor created to execute the code in a stage's boot clause. Periodic usage is observed after time 100, as the sender's behaviour is executed repeatedly.

The 'Native' plot shows the behaviour of the same program, compiled using an Ensemble to C compiler. The memory usage is considerably lower, mainly because the stack size needed for each actor is calculated at compiletime and set in the C code. This is in contrast to the VM, where the stack size of each actor must be large enough to run the interpreter, regardless of which Ensemble application is being run. However, it should be noted that there is still more than enough memory available to run the interpreted program, approximately 50% of the 10 KB RAM on the sensor. Note that the important result here is that the runtime

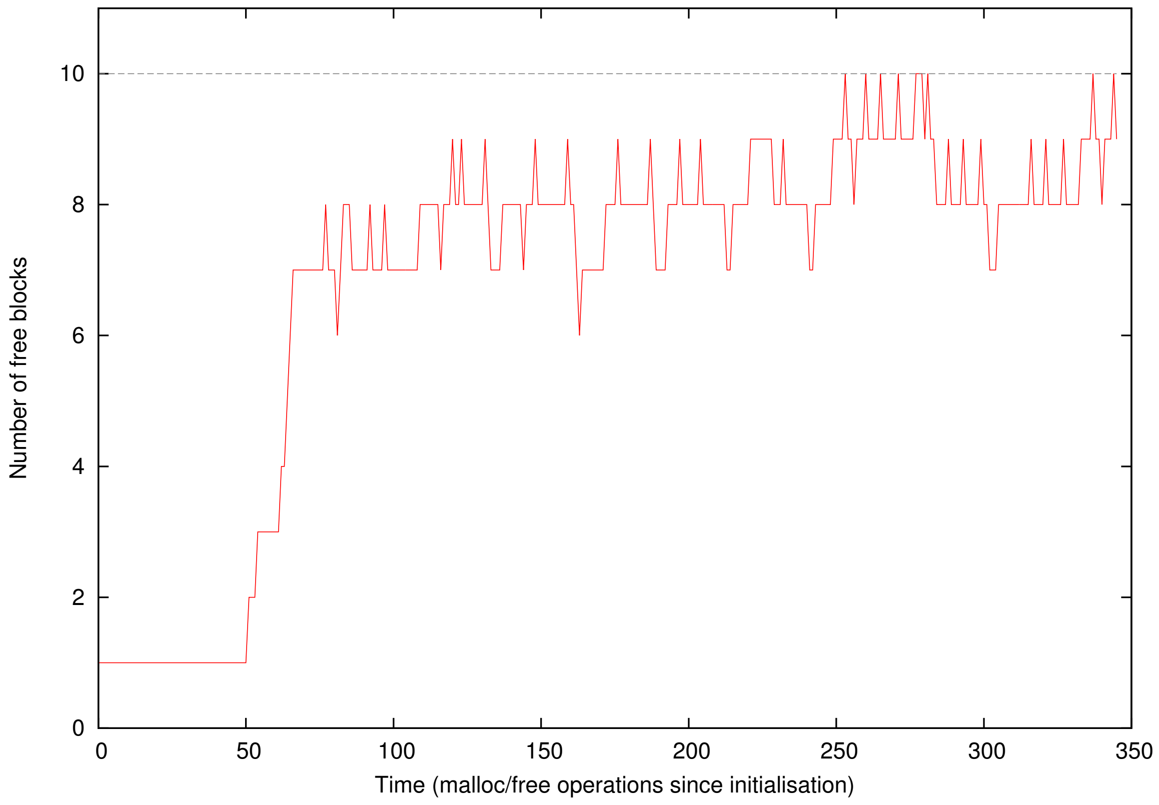


Figure 5.2: External fragmentation: number of free blocks in the interpreted *RadioSense-ToLeds* application.

and application are able to fit within the limited amount of RAM available on the hardware platform. The cost of higher RAM usage, when compared to the native result, is larger power consumption. The ability to measure this was beyond the ability of the author, however, it should be noted that the main consumer of power on such devices is the radio, not the RAM.

As memory is allocated and freed, the region of memory used by the allocator becomes fragmented. Figure 5.2 shows the total number of free blocks, and Figure 5.3 shows the size of the largest free block, in the interpreted *RadioSenseToLeds* program over the same time period as Figure 5.1. Although memory is being allocated and freed throughout this time, the level of fragmentation is bounded, and does not increase to the point where it becomes problematic. Fragmentation is addressed in Section 5.1.6.

Table 5.2 compares the total static memory requirements for the interpreted and native versions of *RadioSenseToLeds*. For the interpreted version, ROM usage includes the linked standard library, the bytecode program, and all C code. Again, the native version is considerably smaller, but there is still enough memory available for the VM and interpreted application.

Interpreted applications consume more memory than their native equivalents, however, the VM is not intended to compete with the memory footprint of native execution. Instead, the

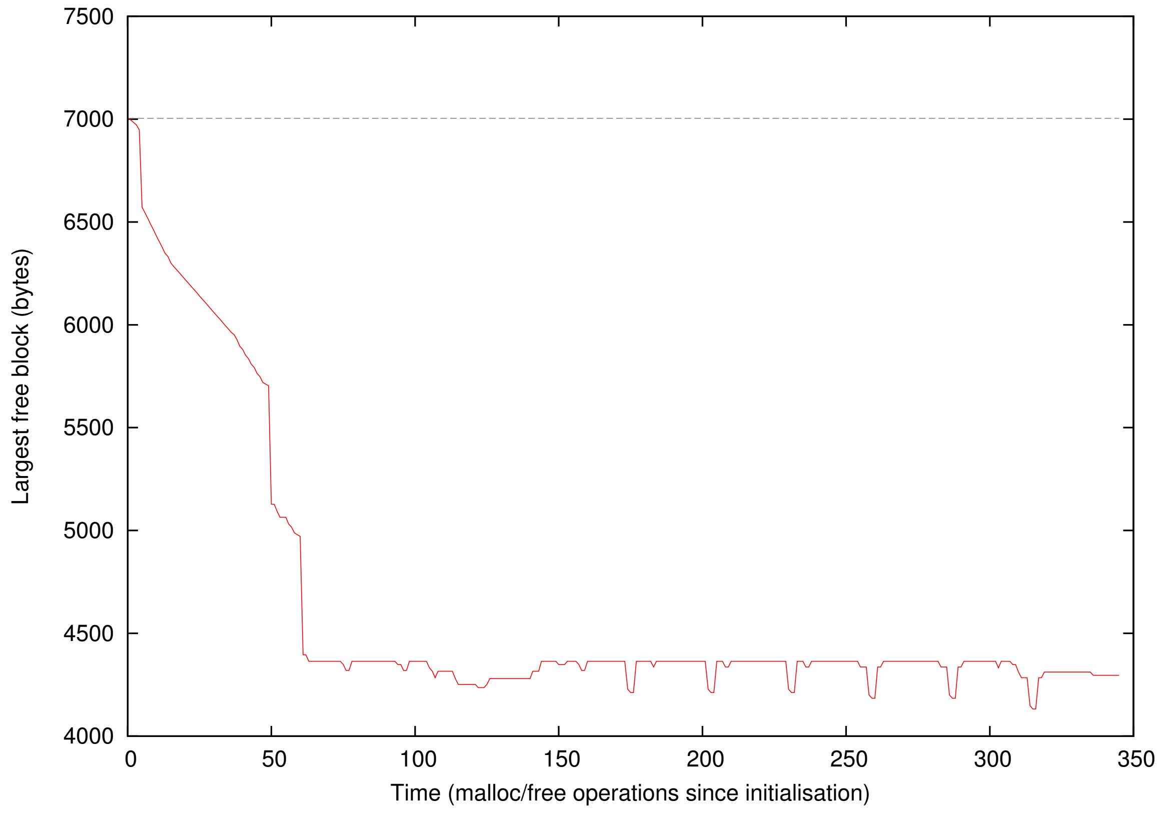


Figure 5.3: External fragmentation: largest free block in the interpreted *RadioSenseToLeds* application.

Type	RAM (bytes)	ROM (bytes)
Interpreted	632	38982
Native	590	22440

Table 5.2: Statically allocated memory for native and interpreted *RadioSenseToLeds*.

VM is intended to make Ensemble more flexible and robust, and to be a base upon which to build useful features such as runtime adaptation. Higher memory consumption is considered an acceptable tradeoff for these features.

DataLogger

Figure 5.4 shows the dynamic memory usage of a more complex interpreted program, *DataLogger*. Between times 500 and 3500, the program is reading from the sensors. The large spikes during this time are buffers filled with data being passed to the flash actor; these are arrays which are copied when they are sent over a channel, in keeping with the strict encapsulation of actors. Between about 4000 and 6000, data is being streamed from flash to the radio. Table 5.3 shows the static memory usage for *DataLogger* and the base station. Despite the complexity of the application, there is still more than 2 kB of RAM and

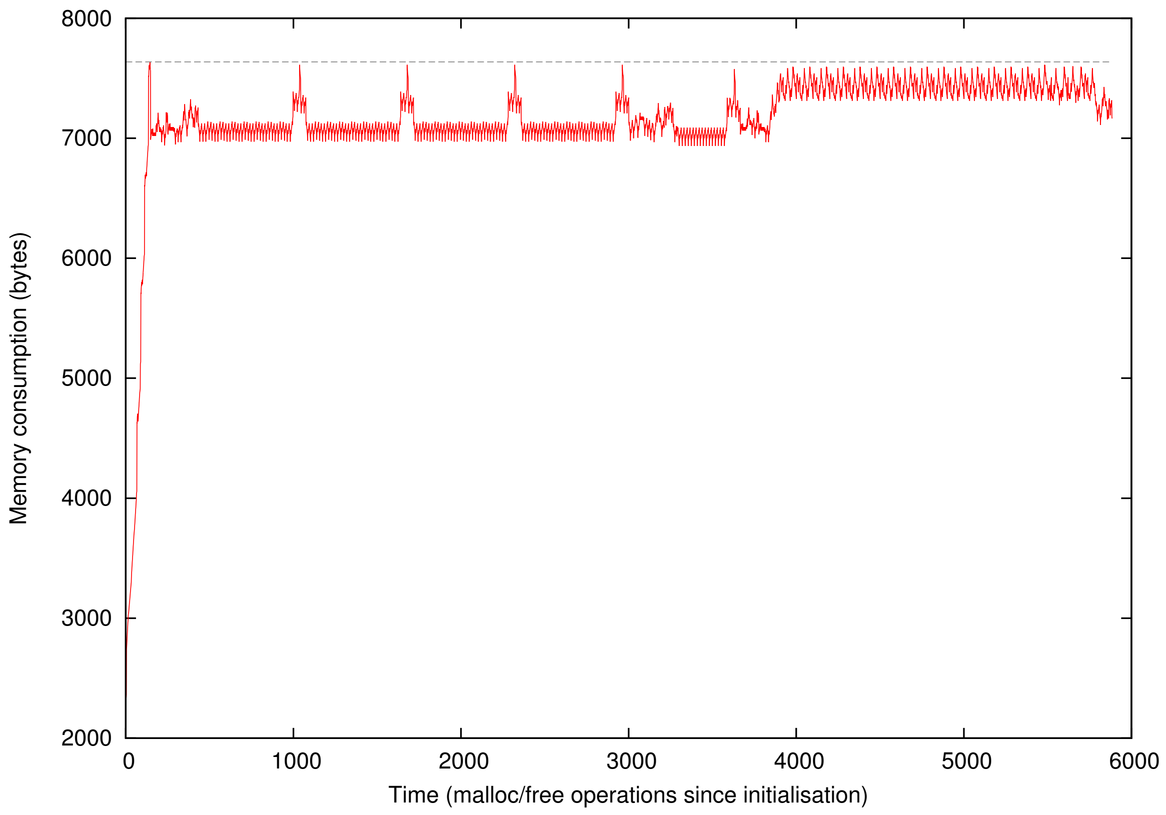


Figure 5.4: Dynamic memory usage of the *DataLogger* application.

Program	RAM (bytes)	ROM (bytes)
DataLogger	636	42866
Base Station	724	41566

Table 5.3: Statically allocated memory for *DataLogger* and the base station.

6 kB of program memory free. This demonstrates that the Ensemble VM is capable of running a realistic and potentially useful application.

5.1.5 Performance

As with Section 5.1.4, the following discussion is in relation to the Ensemble VM. Section C.1.2 has a performance comparison between native Ensemble and nesC code. The following discusses the most relevant results for the VM.

Instructions per Second

One measure of the VM's performance is the number of bytecode instructions executed per second. The VM was instrumented to capture this data, and several programs were run.

Program	Runtime checks	No runtime checks
RadioSenseToLeds	128	132
Fourier	29051	29176
Arithmetic	42894	42871

Table 5.4: Instructions per second.

These measurements were taken on a Tmote Sky sensor node running at 8 MHz. Table 5.4 shows the results.

Different bytecode instructions take different times to execute. The number of instructions a program executes per second depends on which instructions it uses. *Arithmetic* simply performs integer arithmetic in a loop; this is perhaps unrealistic, but it demonstrates the VM's highest speed.

Table 5.4 also shows the effect of runtime error checking on instruction throughput. With runtime error checking enabled, the VM detects various problems as they occur, and throws an appropriate exception. Checks include testing for null pointers, testing for out-of-bounds array accesses, and reporting out-of-memory conditions. If exceptions are not caught by the interpreted program, the offending actor is restarted. With runtime checks disabled, execution continues after an error with undefined behaviour.

As shown in Table 5.4, runtime error checking does not have a significant performance overhead. Thus the only reason to disable error checking is if the memory saved by doing so is needed for bytecode storage.

Execution Time

Counting instructions per second is an artificial measure of performance. A more important measure is the time taken for an application to do something useful; it is unimportant how many instructions are executed in the process. Table 5.5 shows the running times for several applications, both as interpreted applications running on the VM, and as native applications. The figures are the average running times of one hundred executions of each application's behaviour. For each of the interpreted applications, runtime error checking is enabled.

For a computationally intensive task such as the fast Fourier transform, native code is more than an order of magnitude faster than interpreted code. In contrast, for an I/O bound program such as *RadioSenseToLeds*, the overhead of interpretation is negligible.

A sensor node VM would be expected to run mainly I/O bound applications. It is unlikely that computationally intensive tasks, such as the fast Fourier transform, would be written in

Program	Time (s)	Standard Deviation (s)
RadioSenseToLeds (VM)	0.233	0.001
RadioSenseToLeds (Native)	0.230	0.001
Fourier (VM)	2.420	0.001
Fourier (Native)	0.161	0.000

Table 5.5: Running times of interpreted and native programs.

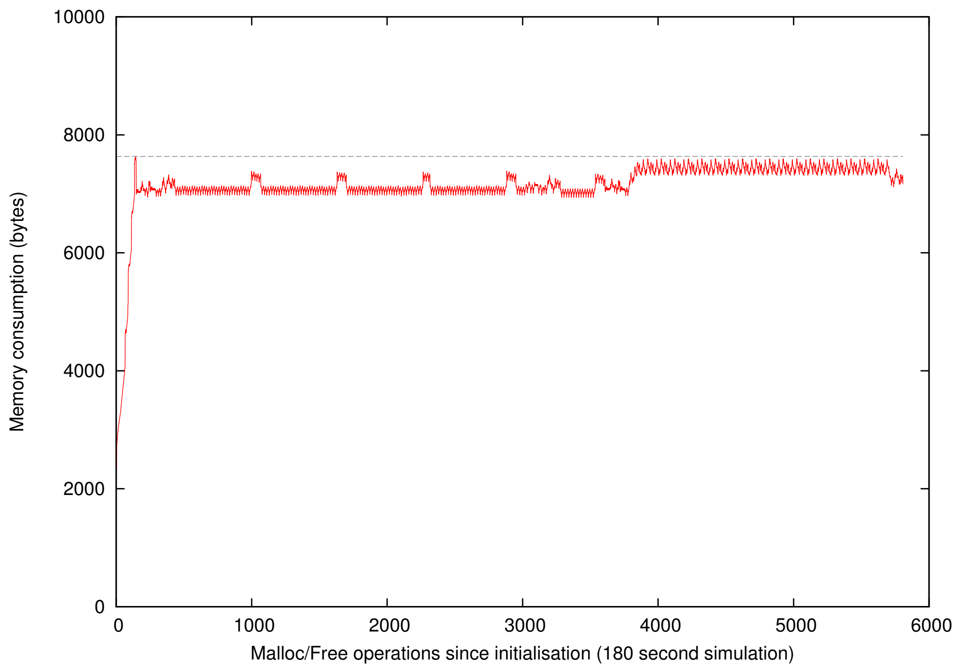
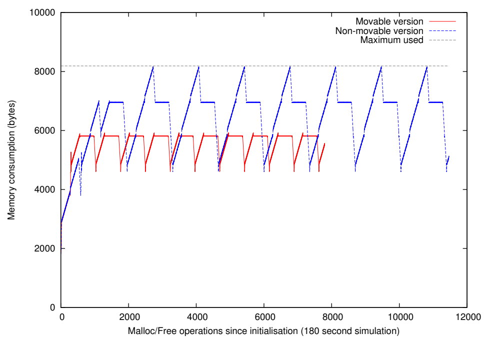


Figure 5.5: Memory usage graph for the movable version of the DataLogger program.

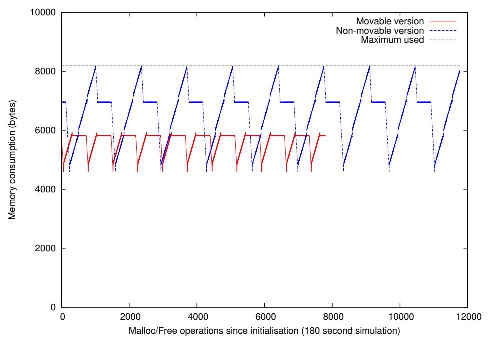
interpreted code. In fact, the C implementation used by the native Fourier example is available to interpreted programs as a native method in the standard library. Similar computationally intensive tasks, where the overhead of interpretation is large, could be implemented in C and exposed to the high-level interpreted program in the same way. This demonstrates that the VM is fast enough for its intended use.

```
type IDataGather is interface(  in bool controller;
                                out bool sas;
                                in mov integer[] sasDone;
                                out integer[] writeChan)
```

Listing 5.1: Modification Required to DataLogger Example for Movability



(a) Memory usage graph for the forwarding example program simulated in Cooja.



(b) Memory usage graph for the forwarding example program on physical Tmote Sky.

Figure 5.6: Memory usage graph for the forwarding example program.

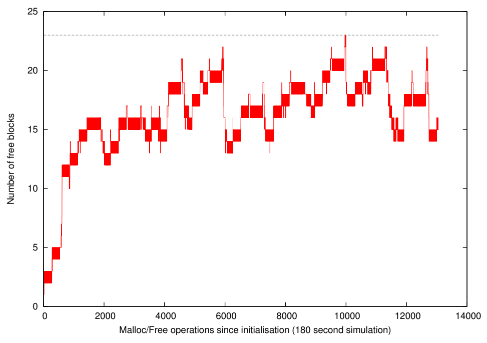
5.1.6 Movability

In order to show the effectiveness of movable types (Section 3.2), the effect of memory allocated from the local and movable heaps was evaluated at the WSN scale against a number of real and contrived Ensemble applications. A comparison is made in terms of absolute memory consumption, memory fragmentation, number of memory allocations, and the largest available block of contiguous memory at a given point in time.

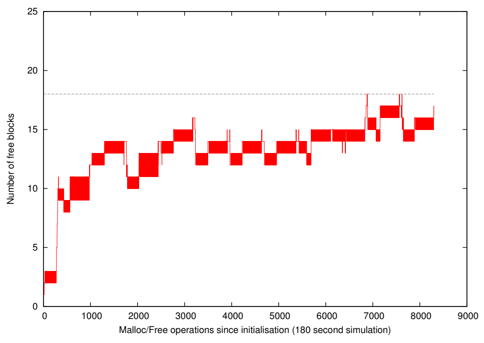
Figure 5.4 shows the memory consumption of the *DataLogger* application using normal memory allocation, whereas Figure 5.5 show the memory consumption for the same application using movable memory. Again, the x-axis shows time in terms of the number of malloc and free operations performed, rather than wall clock time. By comparing the results we see that the spikes observed between allocations 500 to 3500 are less severe in the movable version. This was achieved by marking the `in` channel to the data logger as receiving movable memory, hence, there was no duplication of data when sending data to the storage actor. Listing 5.1 highlights (in yellow) the only change to the *DataLogger* application necessary to provide the efficiency savings of the movable version over the non-movable version.

This application was amenable to the use of movability, requiring only a single addition of the `mov` type to specify that an `in` channel receives movable memory. This removed the need to deep-copy the received data which is subsequently sent on.

The forwarding application (Section 3.2) has been used to generate the results in Figures 5.6, 5.7, and 5.8. These figures show the amount of memory consumed, the number of free memory blocks available, and the largest free block available as the application executes, respectively. For convenience, the latter two figures were generated from data produced by the Cooja simulator. The simulator gives acceptable results when approximating real hardware as shown by the similarity of the results on both real hardware and the simulator, as seen in Figures 5.6a and 5.6b, respectively.

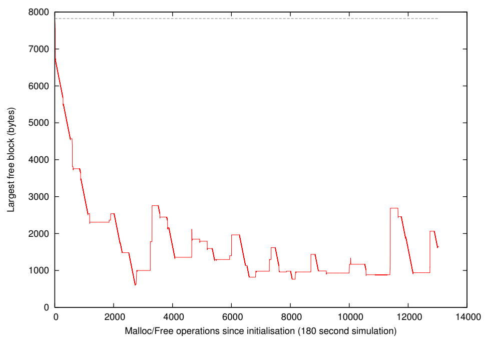


(a) External fragmentation for non-movable version of forwarding application.

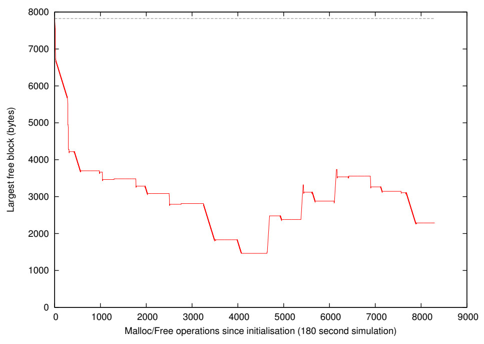


(b) External fragmentation for movable version of forwarding application.

Figure 5.7: External fragmentation for the forwarding example program.



(a) Largest free block over time non-movable version of forwarding application.



(b) Largest free block over time for movable version of forwarding application.

Figure 5.8: Largest free block over time for the forwarding example program.

By examining the results in Figures 5.6a and 5.6b it can be seen that the use of movability has not only reduced the total amount of memory consumed by the application, but also the number of allocations made. Comparing the results in Figures 5.7a and 5.7b shows that movability has reduced the number of free blocks available, and hence the fragmentation in the heap. This is confirmed by Figures 5.8a and 5.8b which show the largest available contiguous memory block at a given point in time for the non-movable and movable versions of the application, respectively.

5.1.7 Real World Deployment

At the time of writing, the Ensemble language and runtime are being used to create a system for the purpose of data collection. This is being done by the Urban Big Data Centre at the University of Glasgow[2]. The application consists of reading values from the light, temperature, and humidity sensors, and then both logging this information locally, and periodically transmitting this information to a central sink node. This sink node will be connected to a desktop machine, which will upload this information to a server. The

[2]www.ubdc.ac.uk/ - Accessed March 2015

application will run on the Tmote Sky hardware platform, and is similar to the *radioCountToLeds*, and the *DataLogger* examples.

The real world deployment will both validate the example applications used to evaluate Ensemble at this level of scale, and when complete will enable an evaluation both of Ensemble *in the wild*, and provide an opportunity to *subjectively* evaluate the practicality, suitability, and usefulness of actors in this application domain for the developers themselves. As the focus of this work has been on the development of the language and runtime, as well as an evaluation from an objective perspective, a subjective evaluation has been beyond the scope of this work.

5.2 The Actor as the Abstraction for Accelerator-based Concurrency

Due to power consumption, heat dissipation, and clock propagation limits, modern hardware architectures are now designed with many, concurrent processing elements, as opposed to single processing elements with increasing clock rates; examples of such architectures include GPUs and multicore CPUs. These hardware platforms are designed to provide the user with multiple physical threads of execution, thus enabling many computations to occur simultaneously.

Software threads have traditionally been used to enable parallel execution on CPU architectures. However, due to the different nature of GPU hardware architectures, a number of different programming techniques are used. OpenCL is a standardised programming framework available for the main GPU vendors (NVIDIA and AMD), as well as other parallel hardware architectures including FPGAs.

While the OpenCL API enables access to these architectures and others, there are three main limitations. Firstly, the user is required to write large amounts of boilerplate code to create the OpenCL environment for a particular calculation. Secondly, the programming style requires explicit data movement between the host CPU and the OpenCL device; this requires manually flattening multi-dimensional arrays and structures of non-primitive types. Thirdly, the language and style used to program the device is often different from the programming language being used on the host. A similar argument can be made against the CUDA framework; since CUDA is only available on NVIDIA hardware, this work is focused on the more broadly applicable OpenCL.

Alternatively, OpenACC is a pragma-based approach to concurrent programming, where a developer explicitly annotates sections of code to be parallelised, as well as the data which should be moved between host and device (Section 2.3.2). While this approach abstracts

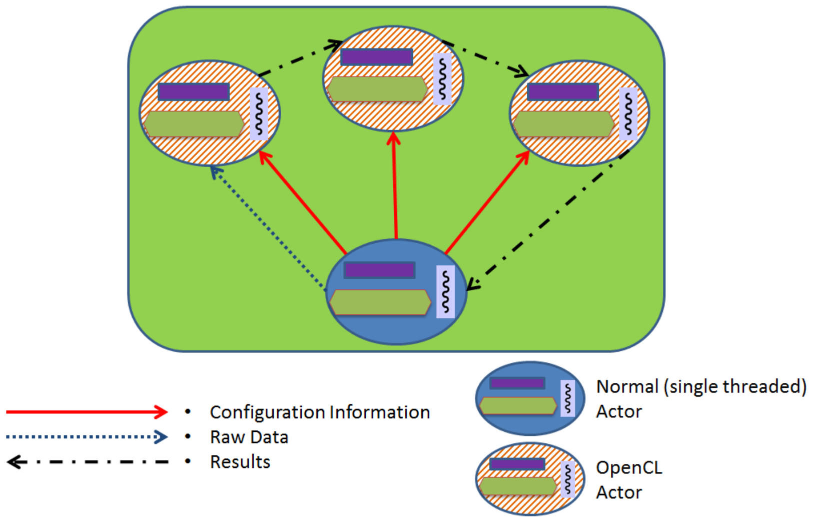


Figure 5.9: Topology of the LUD Application

much of the boilerplate code of OpenCL, applying the simple annotations to single threaded code does not guarantee good performance and is not effective for all application classes.

This section describes the application of actor-based programming to accelerator-based concurrency at the language level by including the OpenCL framework within the Ensemble programming language. The hypothesis is that moving from low-level C code to a concurrent, shared-nothing, high-level actor programming model simplifies the use of OpenCL by providing appropriate structuring, thus enabling greater access to high-performance and heterogeneous computing. This work demonstrates that applications written using actors exhibit less complexity via quantitative metrics compared to handwritten OpenCL in C (C-OpenCL), and that these applications run efficiently on different hardware platforms, with low overhead when compared to C-OpenCL and equivalent or better performance to OpenACC annotated C (C-OpenACC). This is shown for a number of different types of application, including a real-world document ranking example.

5.2.1 Applications

Using the modifications to the language described in Section 3.3, a range of applications were created to evaluate the linguistic complexity and performance of actor-based OpenCL in Ensemble as compared to equivalent C and OpenACC implementations. The applications covered include matrix operations, multiple kernels, parallel reduction and a real world application.

- **Matrix Multiplication** multiplies two 1024 x 1024 matrices in a single kernel.

	Lines of Code			Cyclomatic Complexity			ABC		
Application	C	Ensemble	OpenACC	C	Ensemble	OpenACC	C	Ensemble	OpenACC
Matrix Multiplication	154	-8	5	-1	-2	0	134	2	1
Mandelbrot	96	-4	12	-1	1	0	22	-6	2
Reduction	266	72	3	19	4	1	103	10	0
LUD	144	7	7	5	-8	1	200	-11	0
Document Ranking	45	-16	3	53	-1	0	405	-29	0

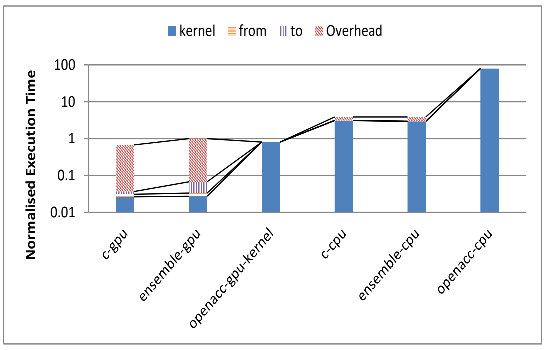
Table 5.6: Difference Between Single Threaded and Concurrent Code per Approach

- **Mandelbrot** computes a 1000 iteration Mandelbrot set in a single kernel.
- **LUD (Lower Upper Decomposition)** factorises a square matrix of 2048 elements, and is a common operation in matrix calculations: this example uses three kernels in series. Figure 5.9 show the topology and connections of the actors in this application.
- **Matrix Reduction** finds the minimal value in an array of 33,554,432 elements using parallel reduction in a single kernel.
- **Document Ranking** takes a set of documents and using a template determines if these documents are wanted, or unwanted: this uses a single kernel.

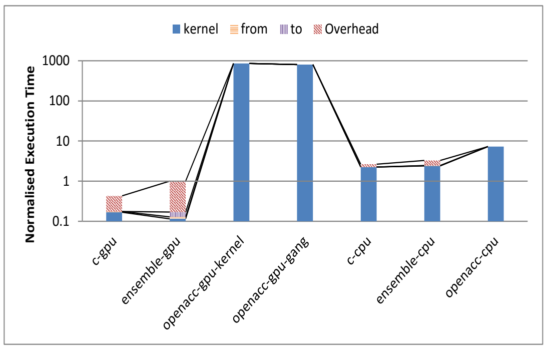
5.2.2 Code Complexity

Table 5.6 shows the (arithmetic) difference between the concurrent and non-concurrent code versions for each approach. For example, the C-OpenCL version of Matrix Multiplication required 154 more lines of code than the single threaded C version, whereas the Ensemble-OpenCL version required 8 fewer lines than the single threaded Ensemble version. As well as the lines of code written, the table also shows McCabe's cyclomatic complexity [136] for the applications. This metric quantitatively assesses the number of different paths through a program. Also shown is the Assignments, Branches, and Conditions (ABC) metric that assesses the size/complexity of code [137]. In each case, the number shown is for the entire application; negative values indicate a decrease in the specified metric.

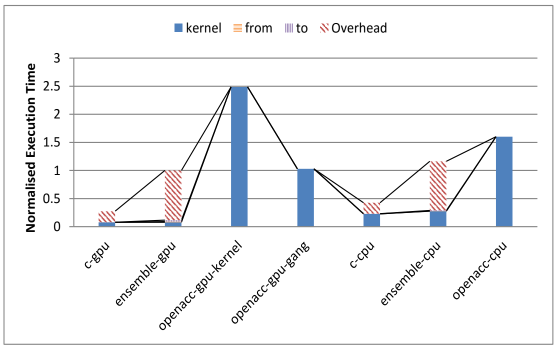
By comparison to C-OpenCL, both Ensemble-OpenCL and C-OpenACC require fewer lines of code, and are simpler by both metrics, with the only exceptions being the matrix multiplication and Mandelbrot cyclomatic complexity examples. The negative values seen are due to the kernel code/actor effectively replacing the outer `for` loop in the single-threaded version, thus reducing code and complexity. Annotating code with OpenACC pragmas generally has little effect on the code size or metrics. The main impact is from having to explicitly specify the sizes of the data to be moved, requiring variables to be accessed.



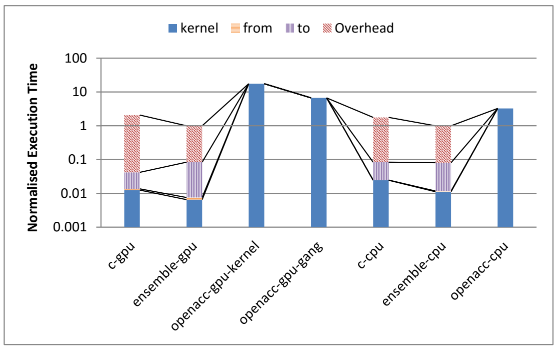
(a) Matrix Multiplication



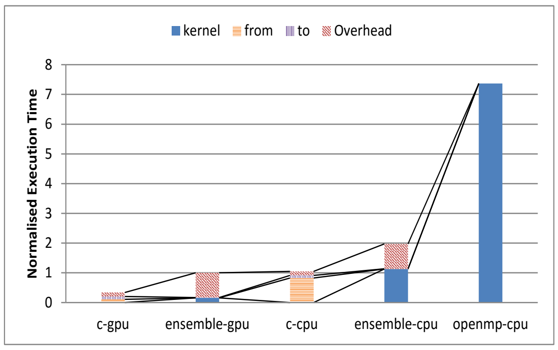
(b) Mandelbrot



(c) LUD



(d) Reduction



(e) Document Ranking

Figure 5.10: Performance between C-OpenCL, Ensemble-OpenCL and C-OpenACC Normalised to Ensemble GPU

The implicit kernel `for` loop, plus the actor and channel abstractions accounts for the better results generally shown by Ensemble-OpenCL compared to C-OpenACC. The main discrepancy being the reduction example, which required very different kernel logic to the single-threaded equivalent in both Ensemble and C, however, this mindset is advocated by both approaches, unlike OpenACC.

5.2.3 Performance

Figures 5.10a-5.10e show the comparative performance of Ensemble-OpenCL, C-OpenCL and C-OpenACC for the applications described above. Here the *CPU* and *GPU* suffix indicates the if the result is for a CPU or GPU, respectively. C-OpenACC refers to both GPU and CPU results, as the PGI compiler was used for both OpenACC and OpenMP pragmas. Each column in each figure displays the time taken to complete the given application, normalised to the Ensemble GPU results. This is done to highlight relative application performance, rather than absolute execution performance. The columns are split to show the relative amount of time taken to move data to a device, move data from a device, to execute a kernel, and the overhead which represents the total relative application execution time minus these values. This does not apply to C-OpenACC as it was not possible to correctly identify the distinct operations due to the `pragma`-based abstraction.

Figures 5.10a and 5.10b show the results for the matrix multiplication and Mandelbrot applications, respectively. The OpenCL actions (data movement and kernel calculations) are nearly equivalent between the C and Ensemble versions. The higher overheads in the Ensemble version are due to interpreter overhead for the non-kernel code. C-OpenACC shows similar performance on the GPU for Figure 5.10a, however much worse performance in Figure 5.10b, even when using the fine-grained `gangs` and `worker` annotations to explicitly specify the `groupsize` and `worksize` to be used on the GPU. Whereas an explicit kernel can take advantage of each thread's position within the 2D architecture of the GPU, the C-OpenACC abstraction cannot, hence the poor performance.

Figure 5.10c is of particular note as it highlights one of the advantages of using channels. In this application there are three kernels, each performing a different operation. In the Ensemble version, a controller actor *plumbs* the channels of the kernel actors together, creating a pipeline of kernels, Figure 5.9. The controller then sends the data, and waits for the final result. By comparison, the C-OpenCL version sequentially invokes each kernel from the host. As C-OpenACC is annotated single-threaded code, it also invokes the relevant code sequentially.

This application highlights the advantage of movability within Ensemble. Without movability, LUD took approximately three minutes (not shown) to complete on the GPU due to all the data movement involved; with movability, it takes approximately five seconds on the GPU. Here we can see that by using movable types, the time taken by Ensemble is comparable to the C version. Again, the higher overheads are caused by the non-OpenCL code being interpreted by the unoptimised Ensemble VM, not data movement.

In order to obtain comparable performance from OpenACC, annotating the outer loop of the relevant code was not sufficient, requiring use of the non-trivial `gangs` and `worker` annotations. This makes the distinction of host and device explicit, with the added disadvantage of being inline in the code, rather than having distinct sections of code. Again, worse performance is seen from the CPU.

Figure 5.10d shows that Ensemble-OpenCL closely tracks the performance of C-OpenCL. C-OpenACC performs poorly for this application on both the GPU and CPU due to this type of application requiring different logic to take advantage of parallel hardware. Again, annotating the sequential version is not enough for this type of application.

Figure 5.10e shows results for a real world example and indicates that the kernel execution time in Ensemble-OpenCL is greater than in C-OpenCL. This is due to some fundamental differences between the host languages, rather than the programming models. Firstly, in Ensemble there are no `NULL` values. This means that all data types must be initialised at the point of creation. This has the advantage of making the language safer, but can lead to increased execution time when the first actual use of a variable it to write to it - i.e. initialisation was not required. In this application, two arrays are initialised in the kernel, within a loop with many iterations. A similar action is taken in the C version, however the two initialisation loops are combined, effectively *halving* the amount of work done by the C version. Loop unrolling in the code generated from the Ensemble compiler could help with this.

Secondly, the ability of C to use an integer value as both a boolean indicator and numerical value leads to faster code when compared to Ensemble, which has no such overloading. Ensemble uses separate types for numeric and boolean values, which in this application requires a number of control structures to be used, and causes greater execution time.

Thirdly, the C-OpenCL version uses short vector types and operations. Ensemble does not yet support OpenCL specific types such as short vectors, and the associated vector operations upon such types. This limitation is due to time constraints, rather than implementation or theoretical barriers. Such types and operations will be added to the Ensemble type system, and will reduce the execution time. Again, this is a limitation of Ensemble, not actors.

The second observation is the smaller data movement time in Ensemble-OpenCL compared to C-OpenCL. This was an unexpected consequence of movability (Section 3.2). In this application, the kernel execution was run multiple times during each individual run to collect sufficiently large time values. In the C-OpenCL version, the data is copied to and from the device each time. No changes or modifications are made to the data between movements. This is also true in the Ensemble version, however due to the lazy evaluation logic the data on the device is never moved back to the host, as it is not required to do so.

The PGI compiler was not able to compile this code, hence no results were obtained for the GPU or CPU from C-OpenACC. The CPU results were generated from the OpenMP `pragmas` and the gcc compiler. Even with the gcc compiler, the results still show slower CPU results by comparison with the other methods.

From the results shown, the general trend is that the performance of hand coded C-OpenCL is comparable to Ensemble-OpenCL. As both C and Ensemble are using the OpenCL runtime, the main difference in time between these approaches comes from the overhead of the Ensemble VM. There are optimisations that can be applied to the implementation, as discussed previously, however the fact that Ensemble is an interpreted language accounts for the majority of the overhead when compared to the C version, as opposed to issues with the language's actor-based structure.

Given these results and the previous discussion, it has been shown that:

- Ensemble-OpenCL always enables simpler, functionally-equivalent code with many fewer lines of code compared to C-OpenCL, and generally simpler, equivalent code with fewer lines when compared to C-OpenACC;
- Ensemble-OpenCL applications present commensurate performance to C-OpenCL on GPU & CPU;
- Relative performance of Ensemble-OpenCL to C-OpenACC ranges from equivalent to significantly better on GPUs, and from better to vastly better on CPUs.

5.3 The Actor as the Abstraction for Adaptive Programming

The combination of improved battery technology and more power-efficient computing hardware has resulted in the proliferation of heterogeneous distributed systems. This *internet of things* consists of embedded devices, wearable devices, hobbiest devices, parallel devices, and commodity devices. Given the different resource and power constraints found in such systems, it is important that applications be able to reconfigure or adapt their runtime execution environment in order to make best use of the resources available.

Adaptable systems can be used in interesting and useful ways for personal, industrial, academic, or educational purposes. The deployment of WSN hardware in hazardous environments [127], makes retrieving hardware to update the software challenging, requiring adaptation or *over-the-air programming* at the firmware level [33], or the module level [46, 38]. At the opposite end of the spectrum, data centres use adaptation to facilitate load-balancing. Virtualisation systems, such as Xen [51], provide coarse-grained adaptation at the OS level. A similar argument can be made for the computing clusters which are used in HPC, often for large scientific computations.

In the consumer domain, industry is beginning to take advantage of adaptable software and the different computers in the surrounding environment. For example, Google's Chromecast[3] enables Android enabled devices to *cast* images or videos to a nearby television or screen, instead of the small screens found on tablets or mobile phones. Apple's

[3]https://www.google.co.uk/chrome/devices/chromecast/index.html - Accessed April 2015

Continuity[4] goes further and lets you move active software between Apple devices for a set of office-focused applications. In both systems, customised hardware is required, and only a specific set of purpose-built applications work.

Despite the potential advantages of adaptable computing, the different categories of hardware often come with domain-specific programming styles and technologies. For example, event-driven programming in embedded systems, and accelerator-based programming in parallel systems, presenting a barrier for adoption of non-experts.

Additionally, there are two main limitations of the above adaptive approaches. The image-based approach requires the replacement of the entire software runtime environment, while the more fine-grained replacement requires the use of APIs. The API-based approach offers no integration with the programming model, requiring the user to have correctly created de-coupled code, and offers no assistance in reasoning about the overall logic of the application or how it will be affected by adaptation.

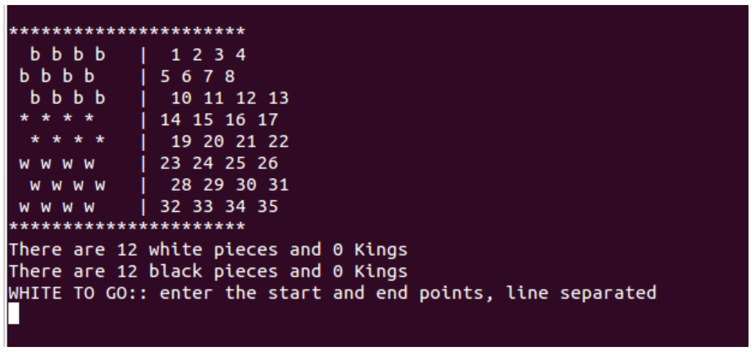
The goal of this work is to support and ease the use of adaptive computing in the general case. Having the actor as the unit of adaptation simplifies the creation of applications which are naturally provisioned for adaptation, in particular runtime migration, without the requirement for specialised libraries, specialist programmer knowledge, or specialist knowledge of the different hardware platforms on which an application may execute, or even the networking technologies which connect them. Providing the ability to adapt actor applications, by default, will make adaptable computing available in the general case, enabling the exploration of programming multi-platform software adaptation. Whereas traditional programming models see distributed systems as distinct hardware locations explicitly interacting with other distinct hardware locations, the actor model abstracts physical locations, enabling users to focus on solving problems.

Given the previous discussion on the design and implementation of the language and runtime in Chapters 3 and 4, respectively, this section describes the application of actor-based computation to adaptive applications. The hypothesis is that the shared-nothing message passing semantics of actors is the appropriate structuring mechanism to create adaptive applications, without restrictive performance costs. This is shown by a combination of raw performance results for the primitive actions of adaptation in the Ensemble language, as well as the performance of two representative applications.

5.3.1 Applications

In order to test the hypothesis that actors are a simple and appropriate unit of abstraction for adaptive applications, two applications were created: an application which enables a human

[4]https://support.apple.com/en-gb/HT204681 - Accessed April 2015



```
*********************
  b b b b   |  1 2 3 4
 b b b b    | 5 6 7 8
  b b b b   |  10 11 12 13
 * * * *    | 14 15 16 17
  * * * *   |  19 20 21 22
 w w w w    | 23 24 25 26
  w w w w   |  28 29 30 31
 w w w w    | 32 33 34 35
*********************
There are 12 white pieces and 0 Kings
There are 12 black pieces and 0 Kings
WHITE TO GO:: enter the start and end points, line separated
```

Figure 5.11: Command Line Draughts Interface

to play draughts against a computer player, and a media player which follows a user around a physical space.

An Adaptive Draughts Engine

One advantage to adaptive computation is the ability to offload data processing to a remote site. Motivation for this includes power constraints, more appropriate hardware, or cost. This idea has been explored via techniques such as RPC/RMI (Section 2.2.5). To show how actors can be used to facilitate offloading, a draughts engine using alpha-beta pruning [138] has been created in Ensemble. The core computation of the application is when the computer player decides on which move to make. To do this, the tree of possible future legal moves is generated from the current board, and the computer player uses heuristics to decide the best move. Generally, the deeper the search tree, the more challenging the computer opponent. The game is played via a command-line interface, Figure 5.11. The application consists of two actor types, one to interact with the user, and one to calculate the computer player's next move.

There are two possibilities for runtime adaptation:

- **spawn** - Before the computer player begins the calculations for its move, it may decide that the calculations are to be performed locally because the game is on easy mode, or remotely, because the game is on hard mode. In the remote case, the computer player searches for a more powerful stage, and spawns an actor remotely, passing the current game board. Once the remote actor has explored the search tree and made a decision, it transmits the choice back to the computer player. This style is similar to the use of RPC/RMI.

- **migrate** - If the computer player has decided that the calculations should be performed locally, but the search is taking longer than some predefined time associated with user response, it may choose to migrate the relevant actors to a different machine to complete the search faster. These stages can either be discovered *a priori* or on demand. Here, the use of migration can transparently enable the existing work to be transferred to another stage(s) to be completed, rather than restarting the calculations, or abandoning the search and returning the best result thus far.

These two approaches highlight that there are different solutions to this problem. In both situations, the partitioning of the application into actors provides the necessary decoupling for either `spawn` or `migrate`. As the language supports these operations natively, their use is trivial from the users perspective.

A Mobile Media Player

This application is designed to show how adaptability can be used in a mobile context. This has been done by creating an application which displays images similar to a slide show. These images will be displayed on different hardware devices as a user moves through an area. As Ensemble does not yet have a graphical environment, these images are ASCII-based. While this is clearly not a modern media example, it represents the salient features of such an application.

The experiment consists of three actors. One actor is responsible for displaying the images (*display*), one actor is responsible for controlling access to the file system (*file*) at the starting stage, and one actor is responsible for locating geographically close stages, and instructing the display actor to either move to the stage closest to the initial stage, or to return to the initial stage if it is about to go out of range (*locator*).

This experiment consists of a laptop and three RaspberryPis, each in a different room. The actors begin on the laptop, and images from the laptop's file system are shown periodically on its display. As the user enters a room, the application will detect the RaspberryPi, and automatically migrate the actor responsible for displaying the image to this device if within a certain distance. This distance is determined using the **#DISTANCE** property for a stage. After this point, the images will be displayed on the RaspberryPi's output. The assumption being that the display of the RaspberryPi is more appropriate. As the user leaves the room, the locator actor will instruct the display actor to migrate back to the laptop. If the user enters another room with a RaspberryPi, the display actor will migrate directly from the previous RasperryPi to the new one.

As the files are located on the laptop, access to them is gained through a file system actor, which accepts paths to files, and returns bytestreams representing images. This interaction is done via channels which transparently *stretch* as the display actor migrates, providing access in the same manner regardless of the stage at which the actor is located. No third party libraries or restructuring of user code beyond that normally expected of the actor-model is required to achieve this adaptation.

This example is a prototype. Given more time, a Java implementation of the Ensemble VM would have been created to enable Ensemble applications to execute on a wide number of mobile phones. Also, the creation of media codecs in Ensemble would enable the application to support more advanced media representations.

5.3.2 Evaluation

Given the previous discussion of how adaptation is represented in Ensemble (Section 3.4) and the limitations or restrictions of adaptation in other actor-based languages in Section 2.1, the following discusses the performance results of using adaptability in Ensemble across heterogeneous hardware platforms. The combination of time constraints and the limitations of other approaches prevented a comparative evaluation. To quantify the cost of adapting actors, the size and time taken to spawn and migrate actors was measured with respect to the applications described above. Also, Appendix C characterises the performance for each adaptation operation in the language in isolation.

The following graphs show *turkey* box plots. The bottom and top of the boxes represent the first and third quartile ranges of the data, with the band inside the box representing the second quartile, or median. The diamond represents the average. The whisker represent the range of values within 1.5 times the inter quartile range, and the circles represent outliers. Each box is generated from 100 runs of the specified experiment; this value was chosen as the results were stable after this many runs.

In the following, the specification of the devices used is discussed in Section 4.3.3.

NXT Platform

Although the NXT platform does support the channel-based abstraction of the network and runtime adaptation, it was not included in the draughts and media player experiments due to a limitation with its communication mechanism. Specifically, the Bluetooth radio chip that it uses attempts to create connections on a restricted Bluetooth channel, hence it cannot create connections to non-NXT platforms. As Ensemble does not use persistent connections and must create connections often, this is a limitation on this platform. It is important to note that this restriction is specific to the NXT hardware platform, and is not an issue with this work.

Given this limitation, the following results are designed to show that it is possible to provide the functionality required for adaptation in a highly resource-constrained environment.

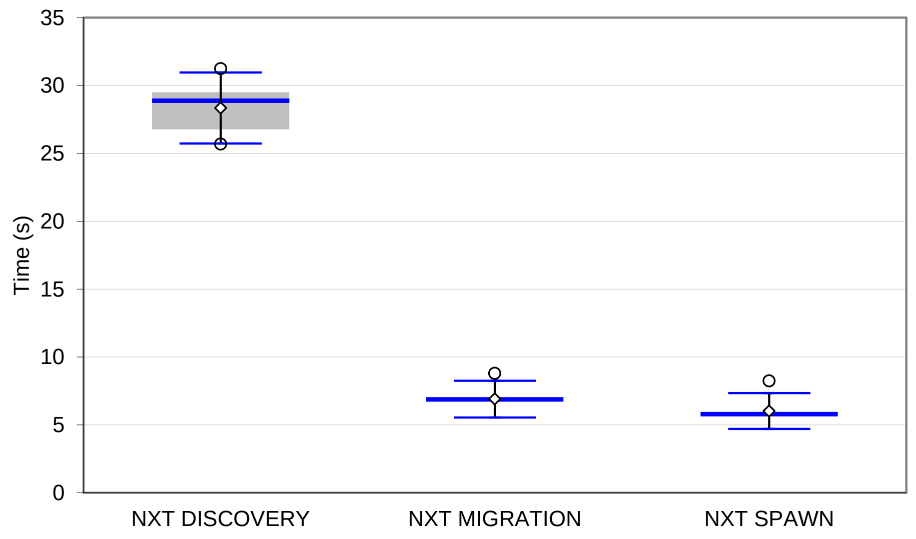


Figure 5.12: Time Taken to perform adaptation on the NXT

Figure 5.12 shows the time taken to discover a stage located on a NXT device, as well as spawn and migrate an actor on to an NXT device from a desktop machine across Bluetooth. The actors used in these experiments contained a mixture of data types, including channels and arrays; although channels with existing connections can not be recreated, the runtime can create remote references, as this is a lazy operation. In this case, the use of the channel generates an exception as the hardware will not be able to create the connection successfully.

The large time seen for discovery is a combination of the seven second timeout for TCP-based connections (of which there are none in this experiment), plus the 10.28 seconds required to discover all possible Bluetooth devices. The remaining 11.07 seconds is required to connect to and communication with the stage on the NXT platform. In general, these times are larger by comparison to the results in the following sections due to the use of Bluetooth, which is unreliable and requires large buffering time periods, and the slower hardware found on the NXT. Considering this, the migration time is expectedly larger than the spawn time. 1540 bytes and 870 bytes were required for migration and spawn, respectively.

5.3.3 Adaptable GPU Programming

As actors are used to represent kernels (Section 3.3), they are naturally able to take advantage of adaptation in the language. The only exception is migration, as explained in Section 4.5.3. Communication with a kernel actor is achieved by channels, hence the performance results in Section C.2.3 are also true for kernel actors.

To show the ease of adaptation of kernel actors, the logic shown in Listing 3.13 was

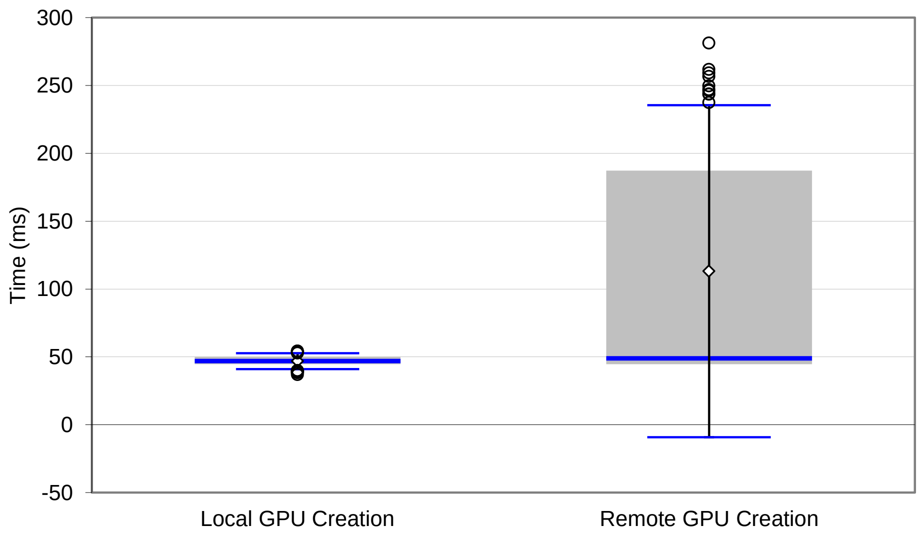


Figure 5.13: Local and Remote Creation Times for a Kernel Actor

modified to create a kernel actor at a remote stage, instead of creating one locally. Listing 5.2 shows the modifications in yellow. It is important to note that all calculations and channel configurations and operations are unchanged. The only changes are the definition of a query, the location of remote stages, and the creation of a kernel actor at that stage.

Figure 5.13 shows the time taken to spawn a kernel actor at a local and remote stage. The experiment was performed between a desktop and a GPU-enabled laptop connected by Ethernet, where actors were spawned on the laptop. On average, local creation takes 46 ms, and remote creation takes 113 ms. The large skew in remote creation is caused by variance in the OpenCL compiler, not the runtime. Although the cost of remote creation is (expectedly) higher than local creation, the average cost is likely to be significantly less than the potential benefit of using a parallel hardware architecture[5]. By constructing applications in this manner, stages without GPU support can easily take advantage of GPU enabled stages.

The next step in this work is to refine the discovery mechanism to provide more fine-grained detail about the types of parallel hardware architectures available, as well as looking at load-balancing using this feature.

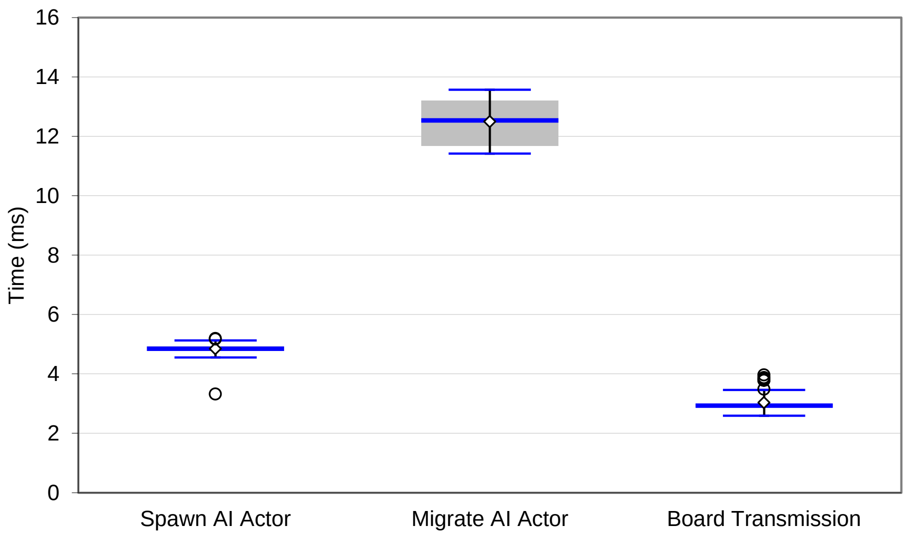
5.3.4 Draughts

Figure 5.14a shows the time to spawn and migrate the actor which is responsible for calculating the computer player's move. The application began on the RaspberryPi, and

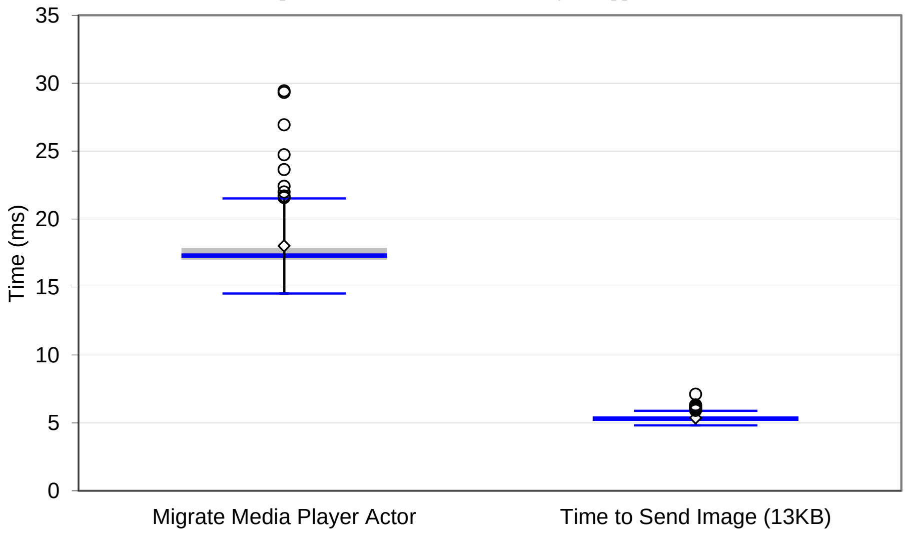
[5]The experiments in Section 5.2 showed performance increases of between 7 and 50 times of parallel code compared to single threaded equivalents

```
...
stage home{
   opencl <device_index=0, device_type=GPU> actor Multiply presents mulI {
      constructor() {}
      behaviour {
          receive req from requests;
          receive d from req.input;
          x = get_global_id(0);
          y = get_global_id(1);
          dim = get_global_size(0);
          c = 0.0;
          for i = 0 .. (dim-1) do {
              c := c + (d.a[y][i]) * (d.b[i][x]);
          }
          d.result[x][y] := c;
          send d.result on req.output;
     }
   }
   actor Dispatch presents dispatchI{

      query gpu_query(){
        $OPENCL == true and $OPENCL_DEVICE == GPU;
      }

      constructor() {}
      behaviour {
          s = 1024;
          ws = new integer[2] of s;
          gs = new integer[2] of 0;
          i = new in data_t;
          o = new out real[][];
          connect dout to i;
          connect o to din;

          ocl_struct = new settings_t(ws,gs,i,o);
          d = generate_data(s);

          stages = findStages(gpu_query());
          m = spawn Multiply() at stages[0];
          connect requests to m.requests;

          send ocl_struct on requests;
          send d on dout;
          receive result from din;
     }
   }
   boot{
     d = new Dispatch();
      <<Remove the kernel actor creation and connection>>
   }
}
```

Listing 5.2: Modifications to Make Matrix Multiplication Adaptable



(a) Adaptation Performance of the Draughts Application



(b) Adaptation Performance of the Mobile Media Player Application

Figure 5.14: Adaptation Performance of the Draughts and Mobile Media Applications

either spawned or migrated an actor to the desktop. It also shows the time taken to transmit a representation of a board, which includes the time taken to marshal and demarshal the data at either stage.

The time taken to migrate is obviously larger than the spawn as the migration must capture the running state of the actor, in addition to the class files transmitted during a spawn. The key point here is that the cost of these operations is not onerous by comparison to the potential benefits of offloading. Given a search depth of three levels, the computers player's move will be calculated on the order of seconds in the opening to mid stages of the game. The cost of adaptation is approximately 100 times faster than the time to complete a search, while the potential benefit is on the order of seconds, depending on the capabilities of the stage being migrated to.

Image	Size (KB)
1	13
2	7.3
3	7.2
4	8.2
5	7
6	9.2

Table 5.7: Media Player Image Sizes

5.3.5 Media Player

In order to be useful as a media player, the migration delay of an actor within the media application must be minimal to prevent the user noticing the transition. For video playback, a succession of images are displayed at a rate of 24 frames every second to provide the illusion of movement. This displays a frame every 41.6 ms, hence the migration time must be less than this. Figure 5.14b shows the time taken to migrate the display actor between a desktop machine and a RaspberryPi, including any images currently referenced by the actor.

The worst case delay is approximately 29 ms, with the average time being approximately 18 milliseconds. This delay includes the time to reconstruct the actor, its state, and all channel connections, meaning that after this time the actor will be able to continue operation.

Six images are used in the media player example; the sizes of each image is shown in Table 5.7. Figure 5.14b also shows the time taken to send the largest image remotely to the migrated actor. The combination of the average migration time (18 ms) with the average transmission time (5 ms) is lower than the 40 ms frame rate delay. It is important to note that this transmission time is less than that seen in Figure C.8 in Appendix C for a similar data size. This is because the latter figure uses a more complex data type which takes longer to marshal/demarshal.

5.4 Summary

Given the large number of heterogeneous computing platforms which are currently available, a number of esoteric programming styles have emerged. The work in this chapter has argued that the actor-model of computation can be used to present a simpler homogeneous programming model both within and across different computing domains, and that executing such a model on a VM does not compromise performance and enables fine-grained adaptation.

By comparison with nesC and TinyOS, the current de-facto choice in WSN programming, Section 5.1 shows that the actor-based model of computation has been shown to be both simpler and as performant on highly resource-constrained embedded hardware, across a number of representative and realistic applications. The execution of Ensemble applications on the Ensemble VM facilitates this without being onerous in either space or time. The use of a movable memory space enables the shared-nothing semantics of the actor model and automated garbage collection, without incurring increased memory consumption and fragmentation.

By noting the parallels between shared-nothing actors, which communicate via explicit message-passing, and accelerator-based computation, which requires explicit data movement, actors have been used as an abstraction for accelerator-based programming. Section 5.2 describes that by marking an actor as a kernel, with its behaviour clause becoming the logic for the kernel, and using channels to convey data between the kernel-actor and other actors, the actor model can completely abstract the large amount of boilerplate code require for OpenCL, and provide a more natural programming model for such computation. The use of movability provides a type-safe way for developers to take advantage of common GPU programming optimisations, while again maintain the encapsulation of actors. The performance of using this approach is comparable to hand written C code using OpenCL.

Finally, given the numerous operating conditions which are presented by heterogeneous hardware platforms connected by different networks types, it is no longer sufficient to think in terms of static software configurations for homogeneous devices. By using the actor as the unit of adaptation, Section 5.3 has described the performance cost of adapting actor-based applications across a set of heterogeneous devices. These results show that this is possible on highly resource-constrained devices, as well as showing minimal cost for medium to highly provisioned devices, including GPUs.

Chapter 6

Conclusions and Future Work

Two major trends in computing hardware in the last decade are the increasing number of processing cores, both in single CPU chips and in dedicated peripheral devices, such as GPUs and co-processors, and the increase in ubiquitous heterogeneous distributed systems. While these advances present significant potential benefits to performance and enable new forms of digital interaction over traditional desktop computing, programming these devices is challenging at best. To aid the use of these platforms for non-experts or non-computing scientists who seek to benefit greatly from these hardware advances, it is essential to provide better programming abstractions and runtime support.

The goal of this dissertation is to provide such support by showing that an actor-based programming abstraction can greatly simplify programming such hardware devices and systems, without incurring any notable performance penalty, enabling developers to focus on solving problems.

6.1 Thesis Statement Revisited

This section revisits the thesis statement presented at the start of the dissertation to assess the impact of the work presented in this dissertation. The thesis is as follows:

> *The use of encapsulated, shared-nothing loci of computation and explicit message passing, found in the actor programming model, will both enable and simplify the programming of concurrent, distributed, and adaptive applications across heterogeneous platforms at different levels of computing scale.*

To prove this assertion, the following work has been done:

Chapter 3 describes the design of an actor-based programming language. The decision to create a new language was due to the limitations of other actor-based approaches. This language supports the creation of applications based on shared-nothing loci of computation which interact via explicit, typed message-passing. As well as automated garbage collection of heap allocated memory, the language supports the idea of a simple movable heap space to address the increased heap usage and fragmentation caused by the use of shared-nothing semantics and automated garbage collection. Furthermore, this chapter describes language support for accelerator-based computation via an actor-based abstraction. This model fits well due to the parallels between these two idioms.

Chapter 4 describes the process of compiling Ensemble applications into a custom class file format, as well as the implementation of a runtime which interprets these applications. The runtime natively supports the concepts expressed in the language, such as actors and channel-based communication, but also the actor-based abstraction and execution of OpenCL kernels, and the discovery and runtime adaptation of actors and stages, as well as a channel-based abstraction of network communication. There is also a discussion of porting the runtime to a number of different hardware platforms, including resource-constrained embedded systems.

The main justification of using an actor-based abstraction for programming concurrent, distributed, embedded, and adaptive applications is made in **Chapter 5**. The chapter is split into three sections:

Firstly, a justification of the actor as the unit of abstraction for embedded programming is made. To show this, a number of applications which covered the different equivalence classes of activity found in embedded applications were used to compare this work with the popular TinyOS/nesC system. In terms of linguistic complexity, Ensemble applications express much simpler, functionally equivalent code when compared to nesC equivalents. Performance comparisons show that when Ensemble is compiled to C code directly, it provides at least equivalent performance to TinyOS, and still had plenty of RAM and ROM available on the embedded hardware. This chapter also discussed the runtime cost of interpreting Ensemble applications by the custom Ensemble VM on resource-constrained hardware. The results show comparable performance for typical embedded applications to the native performance, and do not show excessive resource consumption. Hence, interpreted applications are a valid base to explore runtime adaptation on embedded devices. Also, this section proves that the movable memory space can be used to reduce the increased heap fragmentation and allocation costs incurred by the use of automatic garbage collection and shared-nothing semantics.

Secondly, a justification that using an actor-based abstraction for programming accelerator-based concurrency is made. Specifically, by noting the parallels between the two programming models, an actor is used to abstract the representation of a kernel, and channel-based communication abstracts the explicit data movements between actors, kernel or otherwise. The use of objective software complexity metrics have shown that the actor-based abstraction is significantly simpler when compared to equivalent C and simpler when compared to equivalent OpenACC implementations across a range of applications. Also, performance results show significant improvements when compared to OpenACC, and comparable performance to C. By using the movable heap space, developers are able to leave data on an accelerator (a common optimisation) to improve performance, without violating the shared nothing semantics of the actor model.

Thirdly, a justification of using actors as an abstraction for adaptive computation is made. Given the description in previous chapters of how adaptation, as expressed in Ensemble, overcomes or addresses the limitations in other actor-based languages, this section focuses on performances results. There are performance results for an embedded device communicating over Bluetooth to show that such adaptation is both possible and feasible; limitations in the hardware prevented a full set of experiments. Also, there was reference to a suite of micro benchmarks in Appendix C showing the minimal cost of individual adaptation operations, as well as a discussion and results showing the ease of applying adaptation to kernel-actors and the limited cost of doing so. To motivate the argument of the actor as the unit of adaptation, two applications were created to represent a number of use cases for adaptation: a draughts computer game and a mobile media player. The results showed that little effort was required to apply adaptation to these applications, and that the performance costs for discovery, spawn, or migration were minimal compared to the potential performance improvements that runtime adaptation offered.

6.2 Contributions

This work contributes towards the programming of concurrent, distributed and adaptive systems in the following ways:

- **Actor language support for runtime adaptation**

  In order to test the hypothesis, a new actor-based language was created called Ensemble. As well as shared-nothing semantics and explicit message-passing, the language natively supports location transparent communication and runtime adaptation with appropriate mechanisms for handling both local and distributed failure.

- **The creation of a language-specific VM**

  To compliment the language, a new VM was created, with the specific purpose of executing Ensemble applications. As well as natively supporting the operations expressed in the language, such as adaptation, the VM also executes on a number of different hardware platforms; this includes resource-constrained embedded hardware devices. Also, the runtime supports a channel-based abstraction of two different network technologies.

- **A language-level discovery mechanism based on user-defined properties**

  Unlike other actor systems which have limited or no support for runtime actor discovery, Ensemble enables both actors and stages to be located at runtime based on user-defined properties. This enables users to customise how and when different language entities can be discovered, independent of predefined language or runtime choices.

- **Application of actors to accelerator-based programming at the language level**

  By taking advantage of shared-nothing actors which communicate via message passing, and then applying this to accelerator-based computation, this work has shown not only that actors reduce and simplify the code which has to be written, but also that the performance of such a system is comparable to hand-crafted C-OpenCL.

- **Simple Optimisation of Memory Usage via Movability**

  Through a combination of a single addition to the type systems, and compiletime analysis, this work has shown that the requirements imposed by the shared-nothing semantics of actors and automatic garbage collection need not cause increased consumption and fragmentation of the heap at runtime.

6.3 Future Work

The work described in this dissertation has answered the questions posed by the hypothesis, and in doing so has laid the foundation for significant future work. This section outlines a number of directions to be followed.

6.3.1 Formal Verification

Formal verification is the process of ensuring correct, robust, and reliable software through mathematical analysis of program code. The work in this dissertation has a formal grounding, as the flavour of actors used are strongly influenced by the π-calculus. Consequently, there are two potential research direction: *bi-graphs* and *session types*.

Bigraphs

Bigraphical Reactive Systems (BRS) are a recent formalism for modelling the temporal and spatial evolution of computation. It was initially introduced by Milner [139] to provide a fully graphical model capable of representing both connectivity and locality. A BRS consists of a set of bigraphs and a set of reaction rules, which defines the dynamic evolution of the system by specifying how the set of bigraphs can be reconfigured. The development of bigraphs has been directed toward the modelling of ubiquitous systems by focusing on both mobile connectivity and mobile locality [140, 141]

The hypothesis is that bigraphs can be used to give some guarantees that the use of actor-based adaptation to evolve an application can be done in a safe manner. By creating a new backend for the Ensemble compiler, an Ensemble application can be automatically translated into a bigraphical representation. At this point a developer can approach an expert with a formalisation of their application, and discuss potential issues, without the need to understand formal methods; the developer need only write their application as normal. Given the natural mapping between actors and bigraphs, this code transformation should be tractable. Initial work has already explored this idea, using hand crafted transformations on static applications [142].

Session Types

Session types [143] are data types which enforce patterns of interaction between loci of computation - specifically, the data types which are communicated, as well as the order in which they are sent. This is most commonly applied to distributed applications, where the interaction is embodied by messages sent on a network. Session types have already been applied to several languages including C [144] and Java [145]. Encapsulated actors with explicit message passing are the perfect candidate for session types. Specifically for Ensemble, which already uses typed channels.

This would be achieved by extending the language to include a `session` type. This type would be defined in a similar manner to a `query`, and would contain the protocol of the session. A channel would then be defined in the normal way, using the defined `session` type as the data type that it conveys. An extra phase would then be added within the compiler to ensure that the use of such a channel in an actor's behaviour clause does not violate the protocol in the channel's session. As with movability (Section 3.2), this would all be done at compiletime, and not require any manifestation in the runtime.

The use of session types would provide guarantees that actors are interacting with each other in the manner specified in the session type. This would be useful for replacement

(Section 3.4.4), providing a strong guarantee that the new actor would interact with others in a well defined manner.

6.3.2 JIT Compilation

As shown in Section 5.2.3, the main limiting factor in Ensemble performance is the overhead of host code interpretation. This problem is common in interpreted languages and is overcome with the use of JIT compilation. While the creation of a JIT is not a new research question, the creation of a JIT compiler for resource-constrained platforms in the general case is, and even more so for actor-based systems. Initial work [146] has shown that this is possible for a modified JVM on single node performing numeric computations.

6.3.3 Mobile Phones

The Ensemble VM is not currently supported on mobile phones. This platform is important as it is truly pervasive, and porting the Ensemble VM to such a device would enable research into adaptation in numerous daily situations.

The Ensemble VM is implemented in C. Although such devices do execute C applications, these are not cross compatible between multiple mobile phone models due to certain driver dependencies. Instead, a Java-based version of the VM would be created which would run as a mobile phone *app*. This would enable the VM to execute on multiple mobile phone models, without hard dependences built into the VM implementation. Having the Ensemble VM executed by the JVM would introduce a memory and performance overhead; however, as the resources available on modern mobile phones are greater than that of RaspberryPis, this should not be onerous.

6.3.4 Load Balancing

High Performance Computing Clusters

As Ensemble uses adaptation to both deploy and relocate actors from within the language, the programming of clusters of either homogeneous or heterogeneous machines should be simpler than current practise.

To explore this hypothesis, an existing set of cluster programming applications will be written in Ensemble, and compared to existing implementations in terms of linguistic complexity and performance. Examples include weather simulation [147], ant colony optimisation [148], and n-body simulation [149].

Apart from being desirable in the general case, the simplification of such programming is required as the creators of such applications are often non-computing scientists. Their goals are to solve their problems, not write code.

Distributed Heterogeneous Systems

The work described in this dissertation provides the tools for fine-grained application adaptation at runtime. The next logical step is to use these tools in the most appropriate way.

Load balancing is an obvious starting point. Currently, schedulers in data centres and work stealing algorithms on clusters focus on observed behaviour of running applications to inform decisions about when and where to relocate execution. While this is objective, it completely removes the involvement of the user. As Ensemble exposes these tools to the user directly, it is well placed to enable an exploration of in-language control of adaptation. Equally, it may be useful to express some requirements for an actor or stage that can be used automatically by the runtime to determine if an actor should be relocated to a more appropriate stage.

A more unconventional approach is an on-demand strategy, using the computer hardware embedded in the world around us. Based on the examples of products from Google and Apple (Section 5.3), it is clear that there is a motivation to take advantage of the computing hardware found in the environment in an on-demand fashion. What if a mobile phone became a person's server? Then a hotel room television could continue playing the game that you started on your phone and migrated just before the battery died. Combining generic application migration and ubiquitous computing is an interesting and fruitful area for systems research.

6.3.5 Summary

There are a number future directions that can pursued, based on the foundations provided by this work. As this work is based on the principals of the π-calculus, the rigours of formal methods can be applied. Specifically, bigraphs and session types can be used to ensure that the interaction of actors is correct, and that the runtime adaptation of actors does not lead to higher-level application logic errors. As the language is interpreted, the use of JIT compilers can help alleviate some of the performance overheads. This is particularly interesting in an embedded context. Finally, perhaps the most interesting avenue of future work is to explore the use of runtime adaptation in different contexts. The need to balance computing loads in clusters of high-powered computers is well understood, but current approaches are often automated, excluding the developer. Equally, as computers are increasingly found in the world around us, the ability to create adaptive applications which can take advantage of these computers in an on-demand fashion presents an interesting research direction, in both systems and language research.

Appendix A

Formal Specification of Movability

This appendix describes the formal definitions used in the movability analysis, and represents the formal specification of the approach described in Section 3.2.

This section is split into two sections. The first describes the analysis of how data sent between actors is represented, and the second describes the tracking of data and the variables which alias them.

A.1 Formulating the Move Analysis

This section presents the data flow equations required to track movability of heap objects within an Ensemble application. In particular, we first define the intraprocedural data flow equations for handling movability, abstracting away the handling of aliasing which is subsequently presented as another set of data flow equations in the following section, then we extend the equations to the interprocedural setting using the theory presented earlier in Section 3.2.4.

A.1.1 Intraprocedural Move Analysis

The equations act on a data flow framework defined as $(L, F, \cup)$, where L is the bounded meet semilattice of data flow values, F is the set of flow functions operating on elements of L, and $\cup$ is the meet operator to handle control-flow merge points. This bounded meet semilattice of data flow values is the set of variables which cannot be moved at a specific program point, therefore the greatest element is $\emptyset$ and the least element is the number of reference variables to movable memory in the program, M. Furthermore, $M \leq V$, where V denotes the set of all variables in the program. Recall, that a *use* is defined to be any reference to a variable that does not re-define that variable. If an instruction n is $x = y$, then we can say that n is $use(y)$ ("n contains a use of y").

Next, the flow function f_n must be defined. The flow functions for move analysis must take into consideration the aliases of references, including global, actor member and local variables, and reference formal parameters. For the moment, assume the aliases of all references, and reference formal parameters have been computed for all program points u such that for all references (including reference formal parameters) x, $ALIAS(x, u)$ is the set of references that may be aliases of x upon entry to program point u (see Section A.2); that is, x may reference the same memory location as the variables in $ALIAS(x, u)$. Then our equation can be formulated as:

$$f_n(x) = (x - Kill_n) \cup Gen_n \qquad \text{if } n \text{ is } use(v)\text{, and } v \in x\text{, then generate an error} \tag{A.1}$$

where

$$Gen_n = \begin{cases} \{v\} \cup ALIAS(v, n) & \text{if } n \text{ is a send operation sending } v,\ v \in M \\ \{y\} \cup ALIAS(y, n) & \text{if } n \text{ is a send operation sending } y,\ y \text{ is a reference formal parameter} \\ \emptyset & \text{otherwise} \end{cases} \tag{A.2}$$

$$Kill_n = \begin{cases} \{v\} & \text{if } n \text{ is an assignment } v = r,\ v \in M,\ r \in V - (v \cup ALIAS(v, n)) \\ \{y\} & \text{if } n \text{ is an assignment } y = r,\ y \text{ is a reference formal parameter, } r \in V - (y \cup ALIAS(y, n)) \\ \{v\} & \text{if } n \text{ is a receive operation receiving } v,\ v \in M, \\ \emptyset & \text{otherwise} \end{cases} \tag{A.3}$$

The side-effect of $f_n(x)$ is our error checking on the data flow values reaching program point n. Thus, for $x \in L$ and $v \in x$, if at program point n we send v across a channel, an error should be generated since v has already been moved in some path from the entry node of the CFG under consideration to n.

For each program point n, we associate an In and an Out set denoting the data flow values at the entry and exit of program point n, respectively:

$$In_n = \begin{cases} \emptyset & \text{if } n \text{ is the entry node of the CFG} \\ \bigcup\limits_{p \in pred(n)} Out_p & \text{otherwise} \end{cases}$$

$$Out_n = f_n(In_n)$$

A.1.2 Interprocedural Move Analysis

To extend the move analysis to the interprocedural setting we utilise the call-strings approach described in Section 3.2.4, and extend the data flow equations to qualify data flow values based on the calling contexts. Call and return nodes affect propagation of movability through aliasing of actual and formal parameters, this issue is discussed in Section 3.2.4.

A.2 Alias Analysis

Consider a control flow graph, $G = (N, E)$, with entry node n_{entry}. At some node $n \in N$, two program reference variables, x and y, are *may-aliased* to each other at n if they refer to the same object in *at least one* path from n_{entry} to n. If x and y refer to the same object in all paths from n_{entry} to n, then we say x and y are *must-aliased*. This work focuses on may-alias information.

In order to track all references that could potentially refer to movable memory, a data flow framework for computing aliases is defined. The framework for alias analysis is loosely based on the flow-sensitive *may*-alias analysis work [150]. This approach differs is in the use of the control flow graph rather than the sparse evaluation graph (SEG), and performing interprocedural alias computation using the call-strings approach as opposed to realisable execution paths and alias instances. Our approach has the advantage that the extension to the interprocedural case is simpler than propagating alias instances, requiring minimal changes to the definitions of Gen and $Kill$. While using the SEG would be less computationally expensive than using the CFG, the size of Ensemble procedures, and hence the number of nodes in a CFG, in general, is likely to be very small. The following discusses the intraprocedural and interprocedural cases separately.

A.2.1 Intraprocedural Alias Analysis

We define an alias pair, $< x, y >$ for program reference variables x and y, to denote the possibility that x and y may refer to the same memory location at some program point. The data flow equations for alias analysis act on a data flow framework defined as $(L_{AA}, F_{AA}, \cup)$, where L_{AA} is the meet semilattice of data flow values, F_{AA} is the set of flow functions operating on elements of L_{AA}, and $\cup$ is the meet operator to handle control-flow merge points. The elements of the meet semilattice are sets of alias pairs at a specific program point, therefore our greatest element is $\emptyset$ and our least element is the Cartesian product of the number of variables in the program, $V \times V$.

Following [150], the analysis can be defined using In and Out sets for each program point, noting that only assignment statements to references and call sites modify the sets. Discussion of call sites is deferred, as this is handled by the interprocedural phase of the analysis.

Let $g : L_{AA} \mapsto L_{AA}$ be the flow function for tracking aliases. Then

$$g_n(x) = (x - Kill_n(x)) \cup Gen_n(x) \tag{A.4}$$

where

$$Gen_n(x) = \begin{cases} \{(v, u)\} & \text{if } n \text{ is } v := u,\ u, v \in V \\ \emptyset & \text{otherwise} \end{cases} \tag{A.5}$$

$$Kill_n(x) = \begin{cases} \bigcup\limits_{w \in V} \{(v, w), (w, v)\} & \text{if } n \text{ is } v := u \text{ and } (u, v) \notin x, \text{ where } u, v \in V \\ \emptyset & \text{otherwise} \end{cases} \tag{A.6}$$

Equation A.5 says that upon reaching an assignment or declaration statement, $n : v := u$, the alias pair (v, u) is added to the gen set for program point n. Note, the aliases of v before the assignment do not become aliased to u since $v := u$ is equivalent to a pointer assignment in C. Equation A.6 says that upon reaching a statement, $n : v := u$, the analysis adds all alias pairs involving v to the kill set for program point n. If v is involved in any other alias pair as a prefix, it is replaced by one of its former aliases in the alias pair.

In Section A.1.1 we defined the set $ALIAS$ for each pair, program point n and variable $v \in V$; it is the set of the *access paths* that are *may*-aliased to each heap object at the particular program point. For Ensemble, an access path is an expression combining variable names, subscript operators, and field selectors such that the expression evaluates to a reference. Equation A.4 does not provide the set $ALIAS$, but the set of alias pairs valid at a program point. The move analysis requires the alias mapping, hence $ALIAS$ is defined formally from the set of alias pairs provided by our meet semilattice L_{AA}.

Defintion A.2.1. The set $ALIAS(v, n)$ returns the *transitive closure* of the alias pairs of v upon entry to program point n. Formally, $u \in ALIAS(v, n)$ if and only if there exists alias pairs $(v, y_1), (y_1, y_2), \ldots, (y_n, u) \in x$ at program point n, where $x \in L_{AA}$ and represents the set of data flow values entering program point n, and $v, y_1, y_2, \ldots, y_n, u \in V$.

A.2.2 Interprocedural Alias Analysis

Using the call-strings approach, we extend the alias analysis to handle call and return nodes created for each call site; these nodes generate and kill aliases, between actual and formal reference parameters, respectively. We amend Equations A.5 and A.6 to handle these interprocedural cases. $IntraGen_n$ and $IntraKill_n$ handle the intraprocedural cases described in Section A.2.1. Given actual and formal parameters, $a_i, f_i \in V$ for $1 \leq i \leq n$ ($n \in \mathbb{N}$), and $u, v \in V$, we have:

$$Gen_n(x) = \begin{cases} \bigcup_{i=1}^{n} \{(a_i, f_i)\} & \text{if } n \text{ is a call node for procedure call } p(a_1, a_2, \ldots, a_n), \\ \bigcup_{i=1}^{n} \bigcup_{(f_i,u),(v,f_i) \in x \wedge v \neq u} \{(u, v)\} & \text{if } n \text{ is a return node for procedure call } p(a_1, a_2, \ldots, a_n), \\ \{(v, p(a_1, a_2, \ldots, a_n))\} & \text{if } n \text{ is return statement returning } v \text{ for procedure call } p(a_1, a_2, \ldots, a_n), \\ IntraGen_n(x) & \text{otherwise} \end{cases} \tag{A.7}$$

$$Kill_n(x) = \begin{cases} \bigcup_{i=1}^{n} \bigcup_{w \in V} \{(f_i, w), (w, f_i)\} & \text{if } n \text{ is a return node for procedure call } p(a_1, a_2, \ldots, a_n), \\ IntraKill_n(x) & \text{otherwise} \end{cases} \tag{A.8}$$

Equation A.7 generates alias pairs between actual and formal parameters upon entry to the call node for a procedure call site, propagates any aliases created (to formal parameters) during the analysis of a call upon entry to the return node for a procedure call site, and handles aliasing of returned references with their respective call sites.

Equation A.8 kills all alias pairs involving formal parameters upon entry to the return node for a procedure call site. For cases where there are no parameters to the procedure call, no alias generation or killing is performed.

Appendix B

Class File Description

This appendix describes the layouts of the class files generated from Ensemble applications for the default and embedded cases. It also describes the runtime encoding of Ensemble types used for adaptation.

```
Class {
 // unsigned integer representing the length of this class
 u4  class_length;
 // version of the class
 u4  version_number;
 // length of the constant pool in bytes
 u4  constant_pool_length;
 // the constant pool
 ConstantPool[constant_pool_length];
 // Reference in the constant pool to the name of this class
 u4  name;
 // Reference in the constant pool to the super class;
 // this is 0 only for Object
 u4  superclass;
 // number of bytes to allocate for fields, including those
 // of all superclasses, recursively
 u1  fields_size;
 // number of entries in the dependencies list
 u1  num_dependencies;
 // List of references to the constant pool classes
 // representing the dependencies
 u4[num_dependencies] dep_values;
 // The number of 1-byte entries in the field types array
 u1  num_field_types;
 // char values to decribe the layout of the class field entry
 u1[num_field_types] field_types;
 // the length of this is not needed at runtime
 method[] methods;
}
```

Listing B.1: Default Ensemble Class File Format

```
Class {
  // offset of the superclass definition;
  //this is 0 only for Object
  u2 superclass;
  // number of bytes to allocate for fields,
  // including those of all superclasses, recursively
  u1 fields_size;
  // number of 2-byte entries in the vtable,
  // including those of all superclasses, recursively
  u1 vtable_size;
  // values to initialise all the vtable entries to
  u2[vtable_size] vtable_values;
  // number of fields which are object references
  u1 num_object_fields;
  // indices into the fields array for
  // fields which are objects
  u1[num_object_fields] object_fields_indices;
  // only in subclasses of Component
  [ u1 num_channel_fields];
  // only in subclasses of Component
  [ u1[num_channel_fields] channel_field_indices];
  // the length of this is not needed at runtime.
  // These are all ints
  u2[] named_constants;
  // the length of this is not needed at runtime
  method[] methods;
}
```

Listing B.2: Embedded Ensemble Class File Format

```
slot_size = 4 bytes
// 'uN' means an unsigned integer N bytes long
// offsets are absolute byte positions in the file
// starting at 0

Literal = int   // (slot_size)
        | float // (slot_size*2)
        // (null-terminated char array; 8-bit ASCII, no UTF)
        | string

ConstantPool {
    u1  position;  // the current position in the constant pool
    u1  type;      // the type of the constant pool entry
    Literal value; // the value in the constant pool
}

Method {
  // Number of local variables, in slots.
  // Doubles take up two slots
    u1 num_locals;
    // Number of stack slots
    u1 stack_size;
    // Number of parameters to the method, including 'this',
    // in slots. Needed for virtual invocation.
    u1 num_params;
    // in bytes
    u2 code_len;
    u1[code_len] code;
    u1 num_exception_tables;
    exception_table[num_exception_tables] exceptions;
}

// See the JVM spec for an explanation of these fields.
// They are all absolute offsets here, but used as in the spec.
Exception_table {
    u4 start;
    u4 end;
    u4 handler;
    u4 type;
}
```

Listing B.3: Default Ensemble Class file Auxiliary Structures

```
slot_size = 2 bytes
// 'uN' means an unsigned integer N bytes long
// offsets are absolute byte positions in the file,
// starting at 0

Program {
    u2 version num
    [ u2 main; // only in user programs ]
    literal[] literals;
    class[] classes;
}

Literal = int (slot_size)
        | float (slot_size*2)
        | string (null-terminated char array; only 8-bit ASCII)

Method {
    // Number of local variables, in slots.
    // Doubles take up two slots
    u1 num_locals;
    // Number of stack slots
    u1 stack_size;
    // Number of parameters to the method, including 'this',
    // in slots. Needed for virtual invocation.
    u1 num_params;
    // in bytes
    u2 code_len;
    u1[code_len] code;
    u1 num_exception_tables;
    exception_table[num_exception_tables] exceptions;
}

// See the JVM spec for an explanation of these fields. They
// are all absolute offsets here, but used as in the spec.
Exception_table {
    u2 start;
    u2 end;
    u2 handler;
    u2 type;
}
```

Listing B.4: Embedded Ensemble Class file Auxiliary Structures

Data Type	Encoding
integer	i
string	s
real	r
unsigned integer	u
long	l
byte	b
B	boolean
a	any
](type encoding)	array where number of ’]’ indicate dimension
I(type encoding)}	`in` channel begin and end marker
O(type encoding)}	`out` channel begin and end marker
S(type encodings)’)’	struct begin and end markers
A(type encodings)}	actor begin and end markers
F(type encodings)}	interface begin and end markers
;	top level interface separator

Table B.1: Description of the Runtime Type Encodings

Appendix C

Adaptation Performance Evaluation

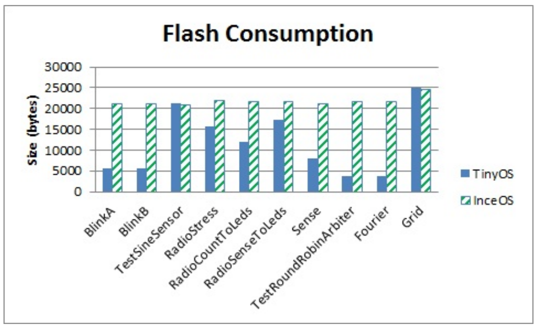
C.1 Native Ensemble Results

This section describes comparison between nesC code executing on TinyOS and Ensemble code executing on InceOS in terms of resource consumption and performance. Unlike the approach described in Chapter 4, here Ensemble applications are compiled to C code directly. This code is then compiled with InceOS to generate a static binary.

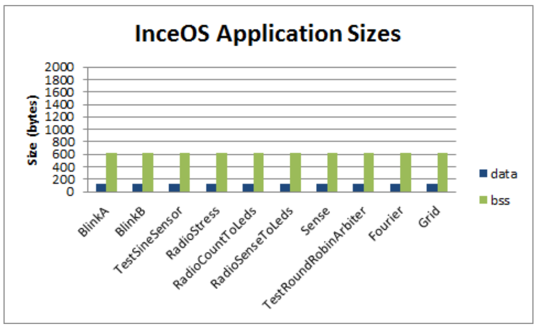
C.1.1 Resource Consumption

Figure C.1a shows the amount of flash which is consumed on the Tmote Sky by TinyOS and InceOS when compiled with an application. Figure C.1b shows the amount of RAM consumed by the data and bss sections of the compiled InceOS and application, and Figure C.1c shows the equivalent for TinyOS. It can be seen that there is variation between the applications on TinyOS compared to the relatively static figures for InceOS. InceOS consumes more flash than TinyOS partially due to the optimisations of the nesC compiler, but mostly due to the extra support mechanisms found in InceOS.

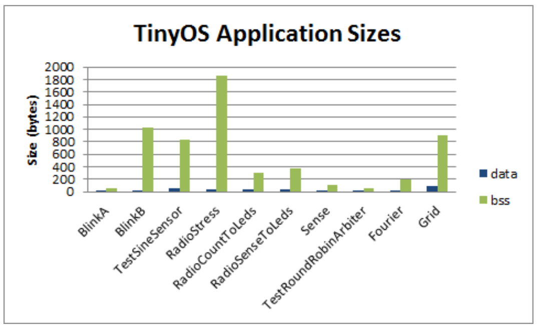
InceOS dynamically allocates the structures used to represent channels and actors at runtime. Dynamic allocation accounts for the small values seen in Figure C.1b, and the lack of conditional compilation causes them to be uniform across the different applications. The use of dynamic allocation and stacks exacts a runtime cost in RAM. The cost for Ensemble and InceOS is highlighted via two examples. The first uses a *Null* actor with no channels and no code in the behaviour section. This null component requires 188 bytes. After the system and *Null* actors are initialised, with both actors and channels being allocated, there are 4363 bytes of RAM available. The second example uses the actor from the Sense example. Here the *sense* actor requires 364 bytes, leaving 4187 bytes of RAM available.



(a) Flash consumed by applications and specified OS.



(b) Space consumed by InceOS applications.



(c) Space consumed by TinyOS applications.

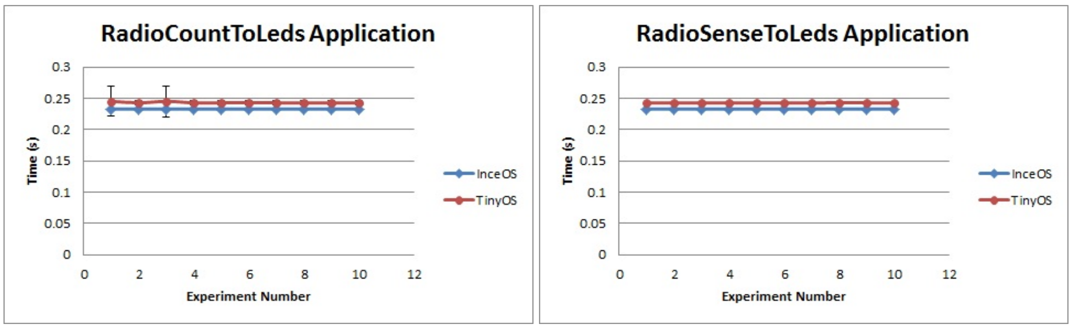
Figure C.1: Comparison of TinyOS and Native Ensemble Size Consumption on Tmote Sky

Taking the sense actor as an example of an average actor, there is enough space on the Tmote Sky to create 11 such actors. 120 bytes are added to the compiler computed stack size for an actor to accommodate system calls and interrupts.

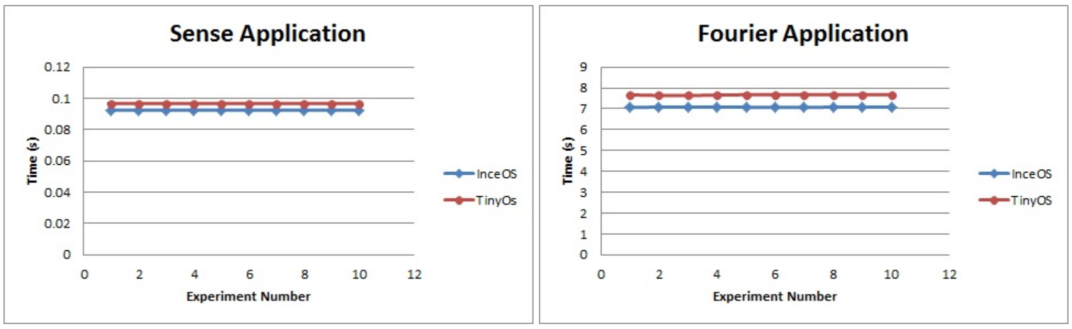
InceOS consumes more flash and RAM than TinyOS, however there is still adequate space available on the motes for even larger and more complex applications; the largest and most complex application in this evaluation, grid, leaves just under 24 KB of flash and nearly 3 KB of RAM available, 50% and 31% of the total space available, respectively.

C.1.2 Performance

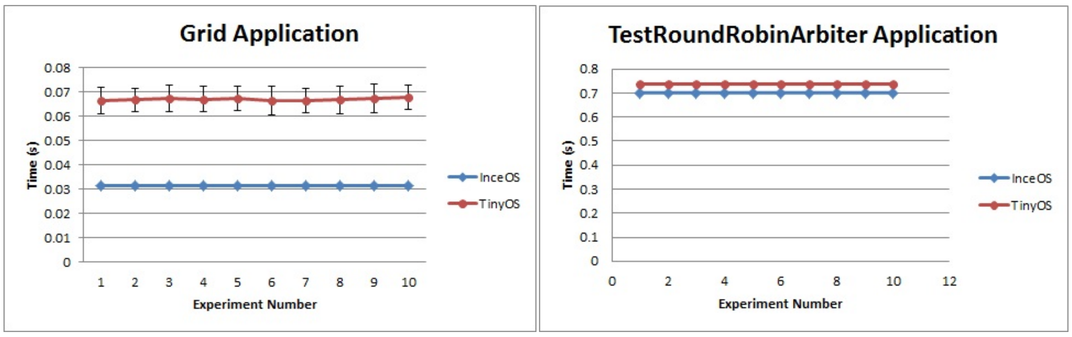
To ascertain if support for the channel and threading mechanisms exact some cost in performance, the performance of a representative cross section of the applications on both TinyOS and InceOS was measured. In the following graphs, each data point is the average of 100 iterations of the application. For example, each point in Figure C.2a is the average time required to increment a software counter and broadcast this value in a packet over the radio 100 times. The error bars on each point represent the standard deviation, however most are not visible as the results are often consistent within the measurement accuracy. Both TinyOS and InceOS are using a csma/ca protocol for radio transmission - i.e., before attempting to send, the radio hardware is queried to detect the presence of other radio transmissions; if radio signals are detected, the transmission is delayed, otherwise the



(a) Comparison of RadioCountToLeds application. (b) Comparison of RadioSenseToLeds application.



(c) Comparison of the Sense application. (d) Comparison of the Fourier application.



(e) Comparison of the Grid application. (f) Comparison of the TestRoundRobinArbiter application.

Figure C.2: Comparison of TinyOS and Native Ensemble Performance on Tmote Sky

packets are sent.

In both Figures C.2a and C.2b, the measurements reflect the sender's action of collecting the data to be sent and sending it. Both figures show a similar performance increase of approximately 4 ms for InceOS as compared to TinyOS. This can be attributed to the fact that the Ensemble behaviour clause is repeatedly executed, rather than in TinyOS where events must be generated before the application can continue to its next iteration. Figure C.2c shows the comparison of the Sense application. We can see that InceOS performs a further 2 ms better than it did in Figures C.2a and C.2b. This is because unlike RadioCount/SenseToLeds, the InceOS Radio actor is not being used (or being scheduled), thus giving more time for the Sense actor to execute.

Figure C.2d shows an example of intense computation; the measurements reflect the time required to complete a fast Fourier transform on a forty element array, and then calculate the maximum. Again InceOS outperforms TinyOS. This is due to the use of tasks in TinyOS to process the Fourier computation. When the task is finished, it must be reposted, requiring an invocation of the scheduler, whereas the behaviour clause in Ensemble is naturally repeated, not requiring any intervention by the scheduler, or extra code from the developer. This highlights that a purely event-driven model is not well-suited to straight computation [129].

The results for the grid application are shown in Figure C.2e. Here the time for a complete iteration of the application is taken: request a slave, give it work and collect the reply. We see that InceOS performs substantially better than TinyOS. The 35 ms difference is caused by a simpler flow through the logic of the Ensemble application and the relatively small number of actions required to access the sensor and radio components. This is in contrast to the disjoint flow necessitated by control switching between the event handlers of the application in nesC, as well as the posting of a task to compute the Fourier transform.

The TestRoundRobinArbiter application results are displayed in Figure C.2f, again showing an InceOS performance gain when compared to TinyOS. Here the 36 ms gain is because the Ensemble channel abstraction is used in InceOS to arbitrate access to the shared resource, whereas additional functionality is required for such arbitration using TinyOS. This particular application is well-suited to the Ensemble blocking channel interaction which naturally handles arbitration.

C.2 Micro Benchmarks

In order to characterise the cost of the different adaptive operations that are provided in Ensemble, performance measurements were made of these operations, and are discussed below. In the following, each experiment was performed on a desktop machine and a RaspberryPi which were connected by Ethernet.

C.2.1 Adaptation of Actors

Null Actors

Figure C.3 shows the cost to adapt a *Null* actor. This refers to an actor with no channels, no state, and the minimum amount of code required to perform the specified adaptation. This test is done to show the cost of the technology, before the addition of developer state, channels, or logic. The results show that in the worst case, 204, 204, and 200 *Null* actors may be remotely spawned with a reference, spawned without a reference, or migrated per second, respectively.

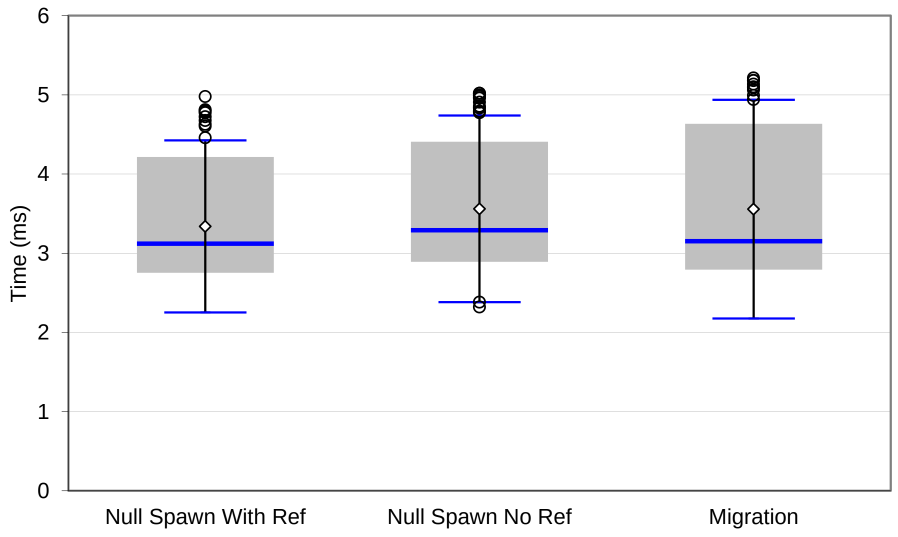


Figure C.3: Adaptation Performance for *Null* Actors

Actors with Varying Quantities

Figures C.4, C.5, and C.6 show the time taken to adapt actors as the number of channels, amount of code, and amount of per actor state changes. Adaptation has been decomposed into spawning an actor without gaining a reference to the new actor, spawning an actor with a reference to the newly created actor, and migrating an actor. Actors were sent from a desktop machine to a RaspberryPi across an ethernet connection. In general, the results show either sub-linear, or linear scaling as the complexity of the independent variable being tested in each case grows.

In these results it can be seen that the increase in code size of an actor has relatively little effect on the median results for each of the three actions in Figure C.4. This is also true as the amount of actor-state changes, Figure C.5. The exceptions being for the 1000 element arrays of integers or structs. As well as the larger amount of data being sent, the data has been fragmented at the network layer, hence the larger times and number of outliers in these cases. As the number of channels grows in Figure C.6, a curve is seen during spawn actions, which follows the pattern expected as the number of channels doubles. For migration, a slightly more noticeable jump is observed for 32 channels.

C.2.2 Discovery of Language Types

Figures C.7a and C.7b shows the time taken to discover a stage or actor respectively. In each case, the number of boolean expressions used in the query was varied to show the cost of simple and complex queries. There were four stages, each with one actor within range which would match the specified query in either case, but the figures show the time for one

Data	Size (bytes)
Integer[1]	21
Integer[10]	57
Integer[100]	417
Integer[1000]	4017
Struct[1]	29
Struct[10]	137
Struct[100]	1217
Struct[1000]	12017

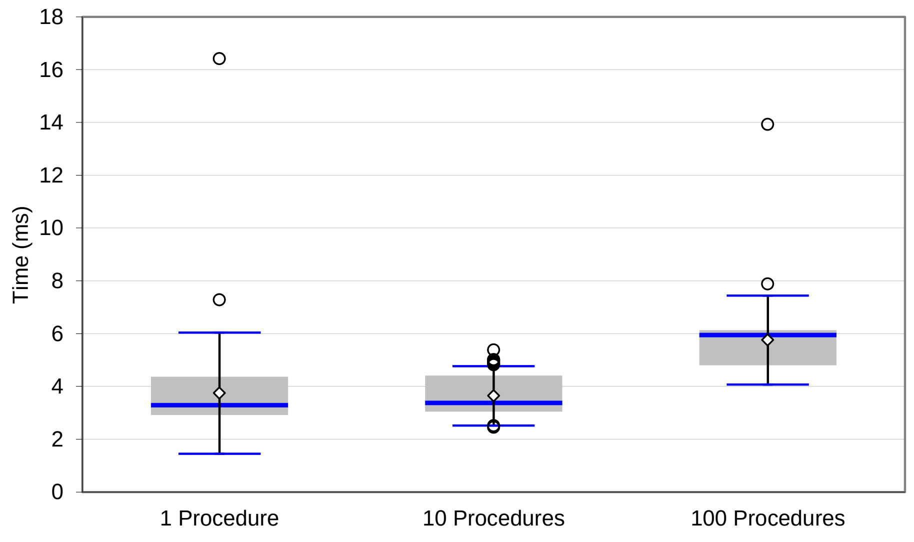
Table C.1: Transmission Size of Different Data Types

to respond. The figures show that the cost of increasing the complexity of a query is almost negligible in terms of performance.

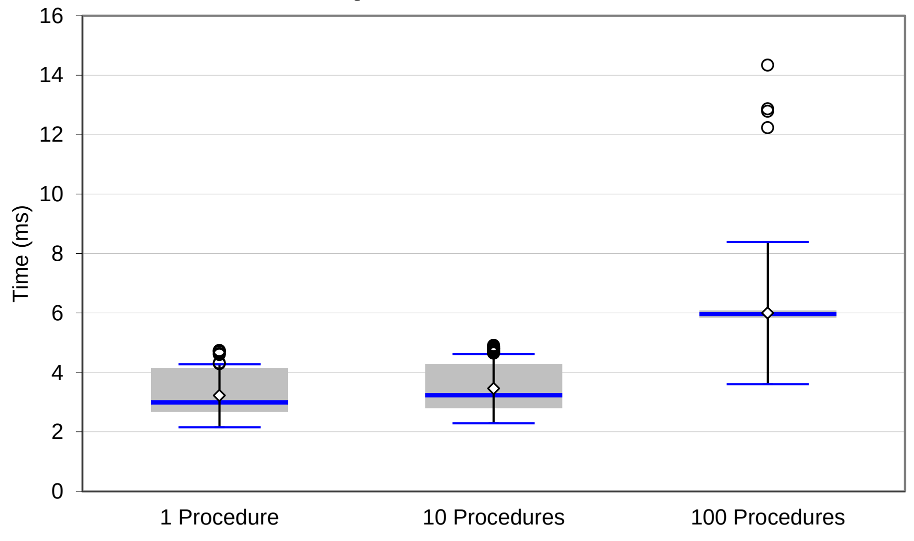
C.2.3 Transmission Times

Figure C.8 show the time taken to transmit different amounts of data between two actors. One is located on a desktop machine, the other is on a RaspberryPi, and the two are connected by Ethernet. Table C.1 shows the size of the transmitted data in each case.

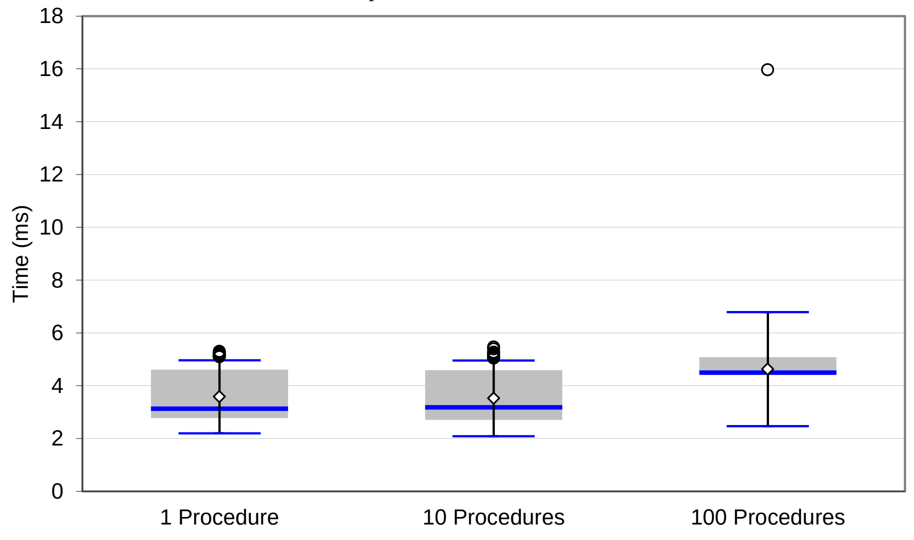
The results shows that the transmission times scale as the size of the data. Again, the large jump for a 1000 array of structures is due to the data being fragmented at the networking layer. This means that the time taken to transmit data is primarily impacted by the performance of the networking layer, and not by overheads in the runtime system or language model.



(a) Time to Spawn an Actor Without a Reference

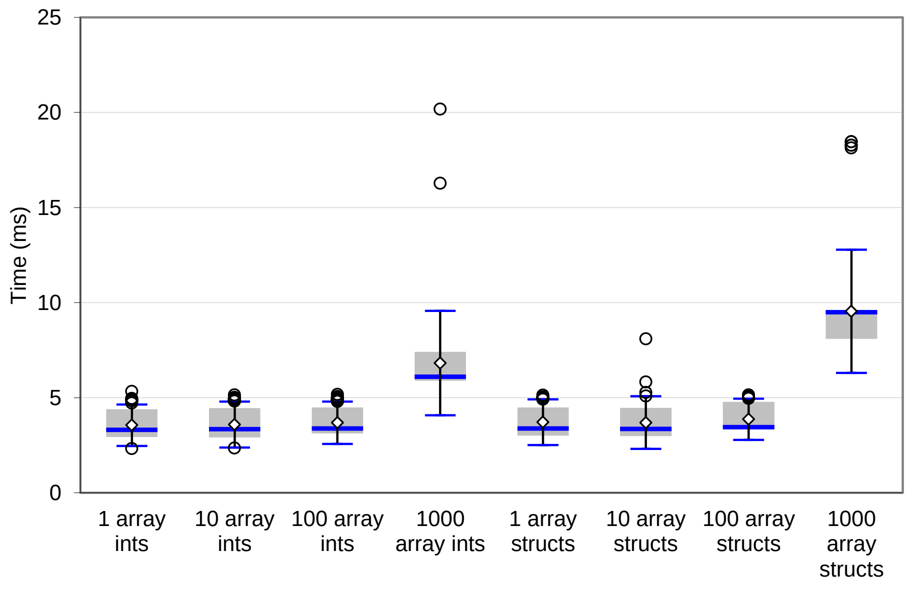


(b) Time to Spawn an Actor With a Reference

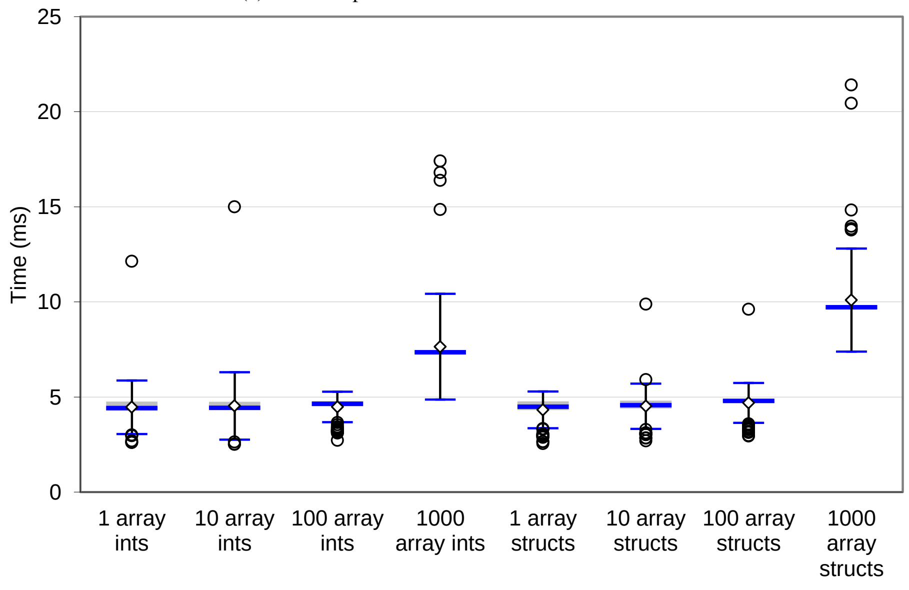


(c) Time to Migrate an Actor

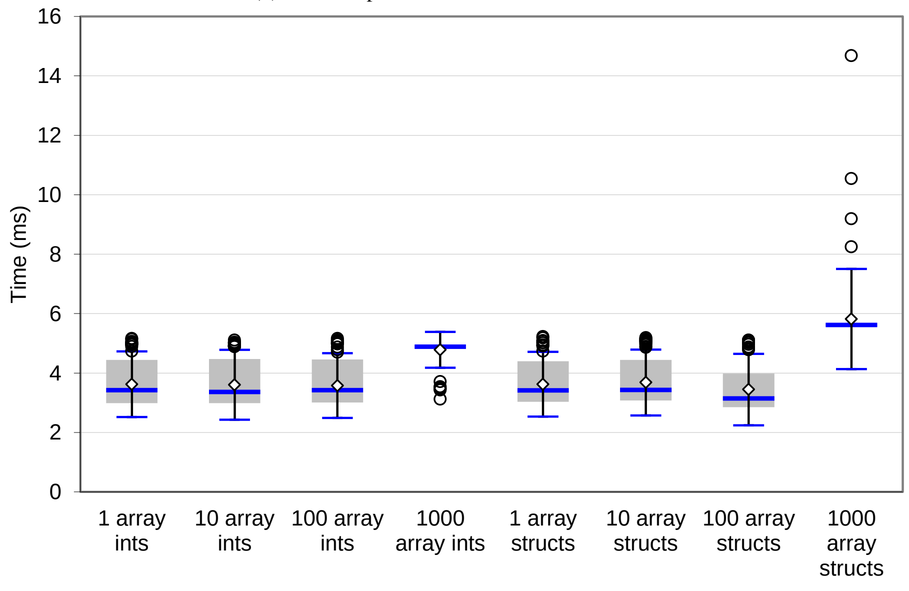
Figure C.4: Adaptation Performance as the Code Size (Number of Procedures) per Actor Changes



(a) Time to Spawn an Actor Without a Reference

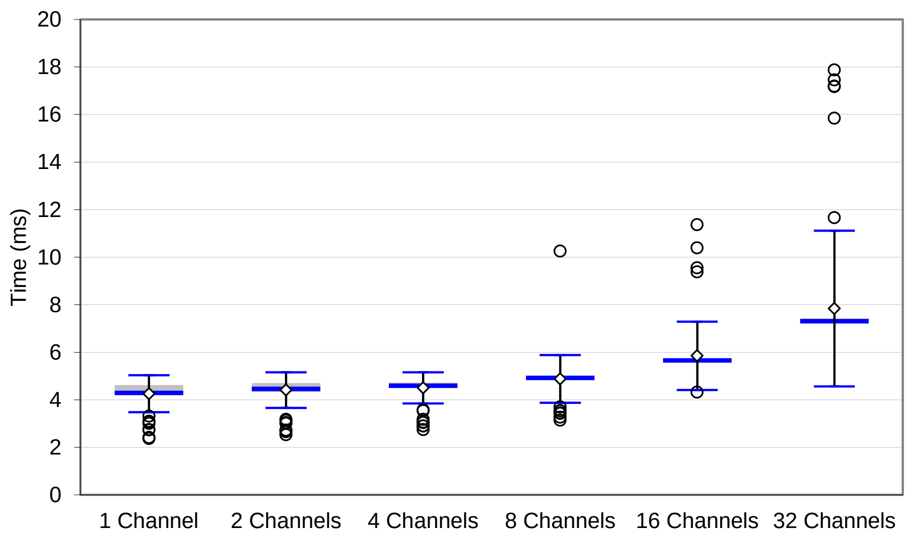


(b) Time to Spawn an Actor With a Reference

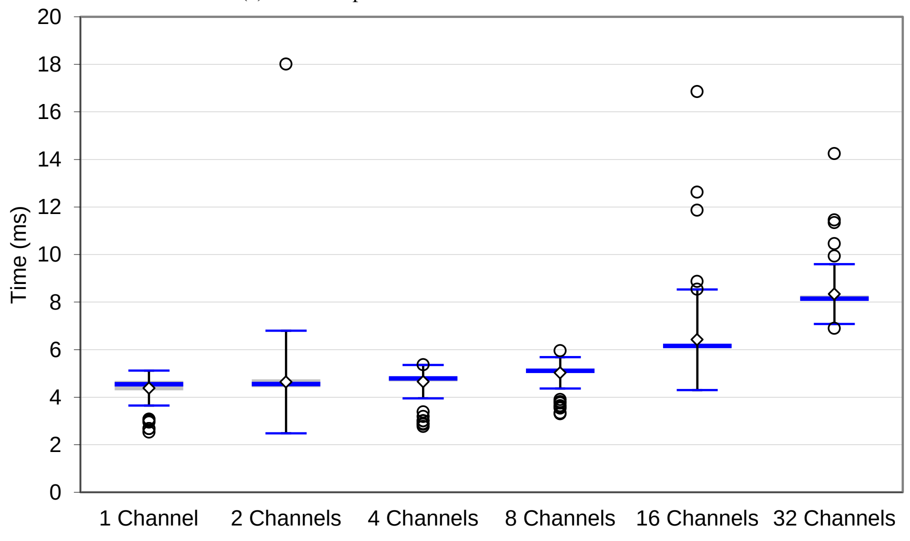


(c) Time to Migrate an Actor

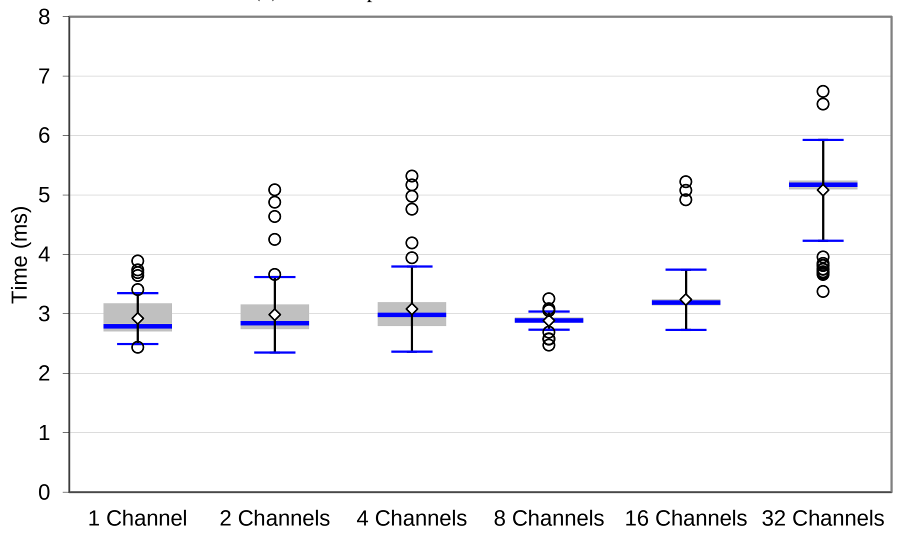
Figure C.5: Adaptation Performance as the Amount of per Actor State Changes



(a) Time to Spawn an Actor Without a Reference

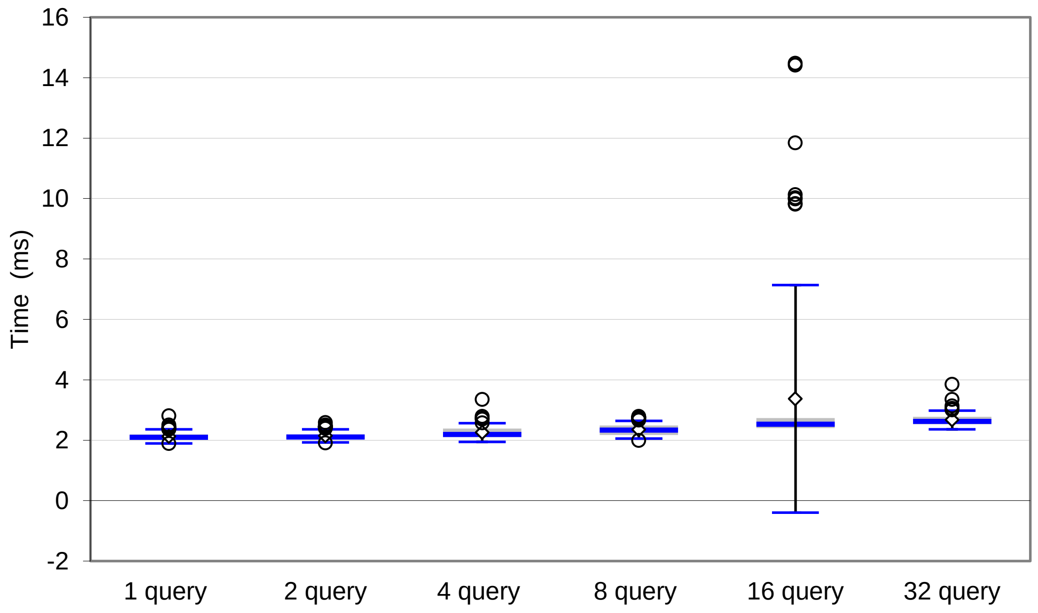


(b) Time to Spawn an Actor With a Reference

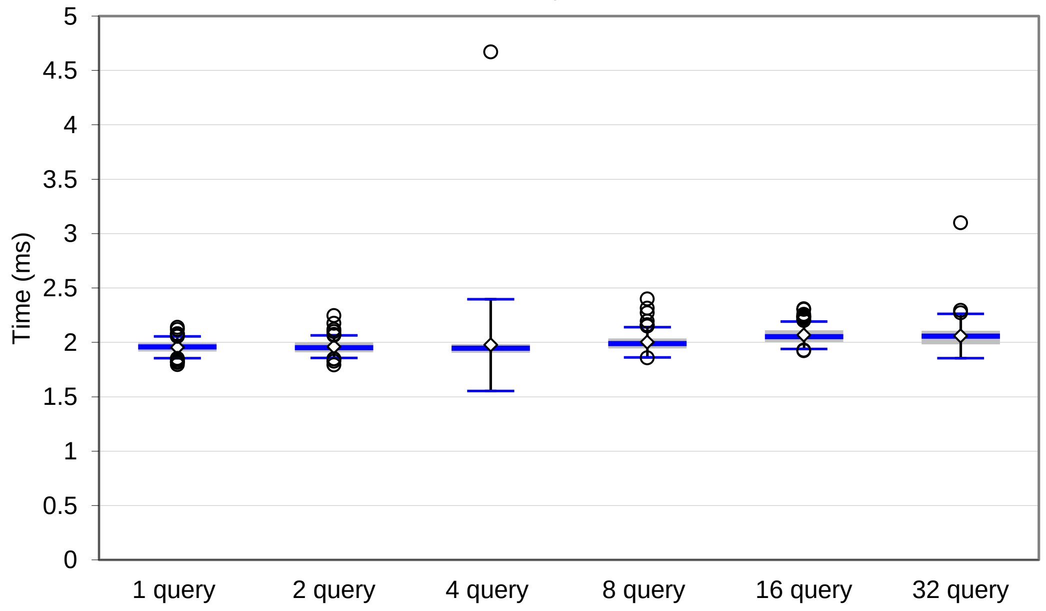


(c) Time to Migrate an Actor

Figure C.6: Adaptation Performance as the Number of Actor Channels Change



(a) Time to Discover a Stage



(b) Time to Discover an Actor

Figure C.7: Time Taken to Discover Actors or Stages as the Query Complexity Varies

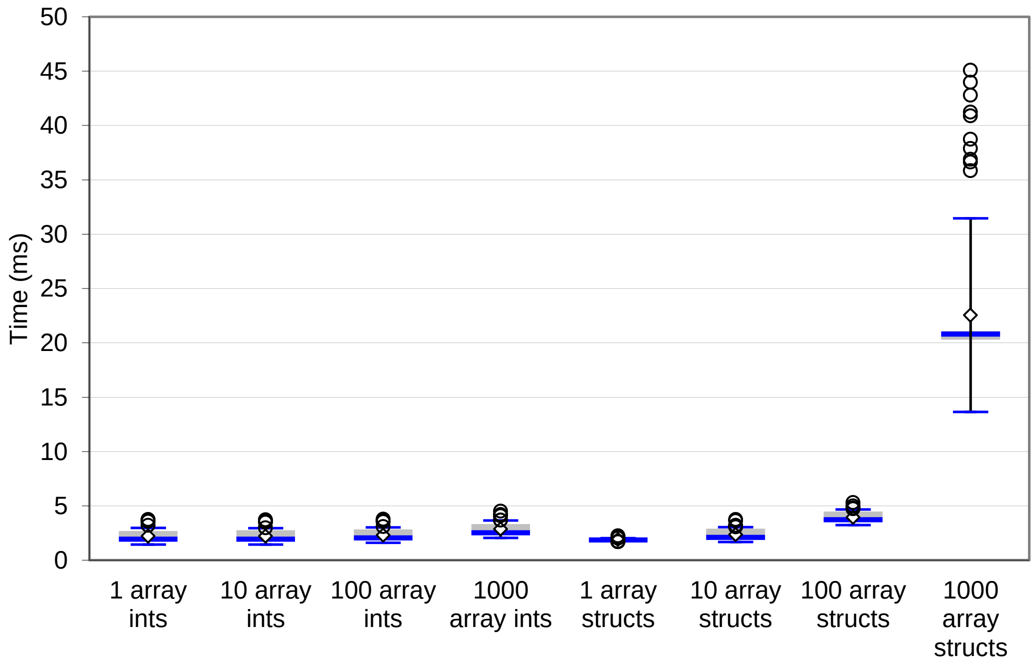


Figure C.8: Time Taken to Transmit Data Between Remote Actors

Bibliography

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The case for a single-chip multiprocessor,” in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII. New York, NY, USA: ACM, 1996, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/237090.237140

[2] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005. [Online]. Available: http://dx.doi.org/10.1109/MM.2005.35

[3] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[4] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” in *Proceedings of the 3rd international joint conference on Artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804

[5] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[6] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, “Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments,” in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH ’13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013.

[7] J. Armstrong, “Making reliable distributed systems in the presence of software errors,” Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.408

[8] H. Svensson and L. ke Fredlund, “Programming distributed Erlang applications: pitfalls and recipes,” in *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007*, S. J. Thompson and L. ke Fredlund, Eds. ACM, 2007, pp. 37–42.

[9] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part i,” *Commun. ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960. [Online]. Available: http://doi.acm.org/10.1145/367177.367199

[10] P. Harvey, “XenoContiki,” University of Glasgow, Department of Computing Science, Tech. Rep., 2009.

[11] S. Conchon and F. Le Fessant, “Jocaml: Mobile agents for objective-caml,” in *Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, ser. ASAMA ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 22–. [Online]. Available: http://dl.acm.org/citation.cfm?id=520788.786423

[12] E. Cheong, J. Liebman, J. Liu, and F. Zhao, “Tinygals: a programming model for event-driven embedded systems,” in *Proceedings of the 2003 ACM symposium on Applied computing*, ser. SAC ’03. New York, NY, USA: ACM, 2003, pp. 698–704. [Online]. Available: http://doi.acm.org/10.1145/952532.952668

[13] E. Cheong and J. Liu, “galsc: A language for event-driven embedded systems,” in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1050–1055. [Online]. Available: http://dx.doi.org/10.1109/DATE.2005.165

[14] P. Haller and M. Odersky, “Event-based programming without inversion of control,” in *Proceedings of the 7th joint conference on Modular Programming Languages*, ser. JMLC’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 4–22. [Online]. Available: http://dx.doi.org/10.1007/11860990_2

[15] M. Odersky and al., “An overview of the Scala programming language,” EPFL Lausanne, Switzerland, Tech. Rep. IC/2004/64, 2004. [Online]. Available: http://lampwww.epfl.ch/~odersky/papers/ScalaOverview.html

[16] J. Ayres, “Implementing stage: the actor based language,” Imperial College London, Tech. Rep., 2007.

[17] C. Varela and G. Agha, “Programming dynamically reconfigurable open systems with SALSA,” *SIGPLAN Not.*, vol. 36, pp. 20–34, December 2001. [Online]. Available: http://doi.acm.org/10.1145/583960.583964

[18] J. Ayres and S. Eisenbach, “Stage: Python with actors,” in *International Workshop on Multicore Software Engineering (IWMSE)*, May 2009. [Online]. Available: http://pubs.doc.ic.ac.uk/actors-in-python/

[19] Z. Porkoláb, J. Mihalicza, and A. Sipos, “Debugging c++ template metaprograms,” in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE ’06. New York, NY, USA: ACM, 2006, pp. 255–264. [Online]. Available: http://doi.acm.org/10.1145/1173706.1173746

[20] A. Dearle, D. Balasubramaniam, J. Lewis, and R. Morrison, “A component-based model and language for wireless sensor network applications,” in *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, ser. COMPSAC ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1303–1308. [Online]. Available: http://dx.doi.org/10.1109/COMPSAC.2008.151

[21] D. Sangiorgi and D. Walker, *PI-Calculus: A Theory of Mobile Processes*. New York, NY, USA: Cambridge University Press, 2001, in-depth walk through of the pi-calculus - very mathematical.

[22] P. Harvey, A. Dearle, J. Lewis, and J. S. Sventek, “Channel and Active Component Abstractions for WSN Programming - A Language Model with Operating System Support.” in *SENSORNETS*, M. van Sinderen, O. Postolache, and C. Benavente-Peces, Eds. SciTePress, 2012, pp. 35–44. [Online]. Available: http://dblp.uni-trier.de/db/conf/sensornets/sensornets2012.html/HarveyDLS12

[23] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy, “The development of the Emerald programming language,” in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 11–1–11–51. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238855

[24] A. Yonezawa, J.-P. Briot, and E. Shibayama, “Object-oriented concurrent programming ABCL/1,” in *Conference proceedings on Object-oriented programming systems, languages and applications*, ser. OOPLSA ’86. New York, NY, USA: ACM, 1986, pp. 258–268. [Online]. Available: http://doi.acm.org/10.1145/28697.28722

[25] A. Yonezawa, “A reflective object oriented concurrent language ABCL/r,” in *Proceedings of the US/Japan Workshop on Parallel Lisp: Languages and Systems*. London, UK: Springer-Verlag, 1990, pp. 254–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=646454.693244

[26] N. Doi, Y. Kodama, and K. Hirose, “An implementation of an operating system kernel using concurrent object-oriented ABCL/c+,” in *on ECOOP ’88 (European Conference on Object-Oriented Programming).* London, UK: Springer-Verlag, 1988, pp. 250–266. [Online]. Available: http://dl.acm.org/citation.cfm?id=60592.60616

[27] C. R. Houck and G. Agha, “Hal: A high-level actor language and its distributed implementation,” in *ICPP (2)*, 1992, pp. 158–165. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.807

[28] R. Panwar, W. Kim, and G. Agha, “Parallel implementations of irregular problems using high-level actor language,” in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS ’96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 857–862. [Online]. Available: http://dl.acm.org/citation.cfm?id=645606.661000

[29] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore, “The CHARM parallel programming language and system:part ii – the runtime system,” *Parallel Programming Laboratory Technical Report #95-03*, 1994.

[30] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for networked sensors,” *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 93–104, 2000.

[31] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, “Mantis OS: an embedded multithreaded operating system for wireless micro sensor platforms,” *Mob. Netw. Appl.*, vol. 10, pp. 563–579, August 2005. [Online]. Available: http://dx.doi.org/10.1145/1160162.1160178

[32] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The *nesC* language: A holistic approach to networked embedded systems,” in *PLDI ’03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, vol. 38. New York, NY, USA: ACM, May 2003, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1145/781131.781133

[33] J. W. Hui and D. Culler, “The dynamic behavior of a data dissemination protocol for network programming at scale,” in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, ser. SenSys ’04. New York, NY, USA: ACM, 2004, pp. 81–94. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031506

[34] J. Jeong, S. Kim, and A. Broad, “Network reprogramming.” University of California at Berkeley, Berkeley, CA, USA, August 2003.

[35] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: a flexible and efficient code update mechanism for sensor networks," in *Proceedings of the Third European conference on Wireless Sensor Networks*, ser. EWSN'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 212–227. [Online]. Available: http://dx.doi.org/10.1007/11669463_17

[36] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, ser. WSNA '03. New York, NY, USA: ACM, 2003, pp. 60–67. [Online]. Available: http://doi.acm.org/10.1145/941350.941359

[37] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462, contiki OS. [Online]. Available: http://dx.doi.org/10.1109/LCN.2004.38

[38] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182810

[39] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, ser. MobiSys '05. New York, NY, USA: ACM, 2005, pp. 163–176. [Online]. Available: http://doi.acm.org/10.1145/1067170.1067188

[40] B. Porter and G. Coulson, "Lorien: a pure dynamic component-based operating system for wireless sensor networks," in *MidSens '09: Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. New York, NY, USA: ACM, 2009, pp. 7–12.

[41] B. Porter, U. Roedig, and G. Coulson, "Type-safe updating for modular WSN software," in *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems*, ser. DCOSS '11. IEEE, 2011.

[42] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1:1–1:42, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1328671.1328672

[43] W. Munawar, M. H. Alizai, O. Landsiedel, and K. Wehrle, “Dynamic TinyOS: Modular and transparent incremental code-updates for sensor networks,” in *ICC’10: Proceedings of the IEEE International Conference on Communications*, Cape Town, South Africa, May 2010. [Online]. Available: http://www.cse.chalmers.se/~olafl/papers/2010-05-munawar-icc-dynamictinyos.pdf

[44] C. Muldoon, G. M. P. O’Hare, M. J. O’Grady, and R. Tynan, “Agent migration and communication in WSNs,” in *Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, ser. PDCAT ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 425–430. [Online]. Available: http://dx.doi.org/10.1109/PDCAT.2008.58

[45] R. B. Smith, B. Horan, J. Daniels, and D. Cleal, “Programming the world with Sun SPOTs,” in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’06. New York, NY, USA: ACM, 2006, pp. 706–707. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176684

[46] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. D. Cid, S. Michiels, C. Huygens, and W. Joosen, “LooCI: a loosely-coupled component infrastructure for networked embedded systems,” in *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, ser. MoMM ’09. New York, NY, USA: ACM, 2009, pp. 195–203. [Online]. Available: http://doi.acm.org/10.1145/1821748.1821787

[47] N. Matthys, R. Afzal, C. Huygens, S. Michiels, W. Joosen, and D. Hughes, “Towards fine-grained and application-centric access control for wireless sensor networks,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC ’10. New York, NY, USA: ACM, 2010, pp. 793–794. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774252

[48] P. Levis and D. Culler, “Mat: a tiny virtual machine for sensor networks,” in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-X. New York, NY, USA: ACM, 2002, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/605397.605407

[49] N. Brouwers, K. Langendoen, and P. Corke, “Darjeeling, a feature-rich VM for the resource poor,” in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys ’09. New York, NY, USA: ACM, 2009, pp. 169–182. [Online]. Available: http://doi.acm.org/10.1145/1644038.1644056

[50] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, "Mote runner: A multi-language virtual machine for small embedded devices," in *Proceedings of the 2009 Third International Conference on Sensor Technologies and Applications*, ser. SENSORCOMM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 117–125. [Online]. Available: http://dx.doi.org/10.1109/SENSORCOMM.2009.27

[51] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177, i have reviewd this from master - go find there. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[52] E. Correia, "A virtual solution to a real problem: Vmware," White Paper, School of Computing, Christchurch Polytechnic Institute, NZ., Tech. Rep., 1998.

[53] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251203.1251223

[54] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft, "Turning down the lamp: software specialisation for the cloud," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863114

[55] I. Giurgiu, O. Riva, and G. Alonso, "Dynamic software deployment from clouds to mobile devices," in *Middleware*, 2012, pp. 394–414.

[56] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: distributed applications through software modularization," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 1–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1516124.1516126

[57] M. P. Forum, "MPI: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[58] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine, "Kanor: A declarative language for explicit communication," in *Proceedings of the*

13th International Conference on Practical Aspects of Declarative Languages, ser. PADL'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 190–204. [Online]. Available: http://dl.acm.org/citation.cfm?id=1946313.1946335

[59] C. H. Moore, "Forth: A new way to program a minicomputer," *Astron. Astrophys Suppl*, vol. 15, pp. 497–511, 1974.

[60] E. D. Rather and C. H. Moore, "The FORTH approach to operating systems," in *Proceedings of the 1976 annual conference*, ser. ACM '76. New York, NY, USA: ACM, 1976, pp. 233–240. [Online]. Available: http://doi.acm.org/10.1145/800191.805586

[61] E. Shein, "Python for beginners," *Commun. ACM*, vol. 58, no. 3, pp. 19–21, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2716560

[62] A. Kennedy and D. Syme, "Design and implementation of generics for the .NET common language runtime," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/378795.378797

[63] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder, "Characteristics of dynamic JVM languages," in *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL '13. New York, NY, USA: ACM, 2013, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/2542142.2542144

[64] N. Costa, A. Pereira, and C. Serodio, "A Java software stack for resource poor sensor nodes: Towards peer-to-peer jini," in *Embedded and Multimedia Computing, 2009. EM-Com 2009. 4th International Conference on*, 2009, pp. 1–6.

[65] J. Boldt, "The Common Object Request Broker: Architecture and specification," Object Management Group, Specification formal/97-02-25, July 1995. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/97-02-25

[66] R. Klauck and M. Kirsche, "Enhanced DNS message compression - optimizing mDNS/DNS-SD for the use in 6lowpans," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, March 2013, pp. 596–601.

[67] ——, "Bonjour contiki: A case study of a DNS-based discovery service for the internet of things," in *Ad-hoc, Mobile, and Wireless Networks*, ser. Lecture Notes in Computer Science, X.-Y. Li, S. Papavassiliou, and S. Ruehrup, Eds. Springer

Berlin Heidelberg, 2012, vol. 7363, pp. 316–329. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31638-8_24

[68] T. Chothia, D. Duggan, and J. Vitek, "Type-based distributed access control," in *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, June 2003, pp. 170–184.

[69] D. S. Milo▮ičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, no. 3, pp. 241–299, Sep. 2000. [Online]. Available: http://doi.acm.org/10.1145/367701.367728

[70] P. Smith and N. C. Hutchinson, "Heterogeneous process migration: The Tui system," Vancouver, BC, Canada, Canada, Tech. Rep., 1996.

[71] J. M. Smith, "A survey of process migration mechanisms," Columbia University, New York, Tech. Rep., 1988. [Online]. Available: http://www.cis.upenn.edu/~jms/svy-pm.pdf

[72] D. Cheriton, "Binary emulation of UNIX using the V Kernel," in *Proceedings of the Summer USENIX Conference*, 1990, pp. 73–86.

[73] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," 1986, pp. 93–112.

[74] H. Peine and T. Stolpmann, "The architecture of the ara platform for mobile agents," in *Proceedings of the First International Workshop on Mobile Agents*, ser. MA '97. London, UK, UK: Springer-Verlag, 1997, pp. 50–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=647627.732412

[75] M. Ranganathan, A. Acharya, S. D. Sharma, and J. Saltz, "Network-aware mobile programs," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '97. Berkeley, CA, USA: USENIX Association, 1997, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268680.1268687

[76] S. Funfrocken, "Transparent migration of Java-based mobile agents: Capturing and re-establishing the state of Java programs," *Personal Technologies*, vol. 2, pp. 109–116, 1998, 10.1007/BF01324941. [Online]. Available: http://dx.doi.org/10.1007/BF01324941

[77] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser, "Mole : Concepts of a mobile agent system," *World Wide Web*, vol. 1, no. 3, pp. 123–137, Mar. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1019211714301

[78] J. Waldo, “Remote procedure calls and Java remote method invocation,” *IEEE Concurrency*, vol. 6, no. 3, pp. 5–7, Jul. 1998. [Online]. Available: http://dx.doi.org/10.1109/4434.708248

[79] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[80] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation,” *Parallel Comput.*, vol. 38, no. 3, pp. 157–174, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.09.001

[81] A. Munshi *et al.*, “The OpenCl specification,” *Khronos OpenCL Working Group*, vol. 1, pp. l1–15, 2009.

[82] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[83] S. Wienke, P. Springer, C. Terboven, and D. an Mey, “Openacc: First experiences with real-world applications,” in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870. [Online]. Available: http://dl.acm.org/citation.cfm?id=2402522

[84] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008, vol. 10.

[85] T. D. Han and T. S. Abdelrahman, “hiCUDA: High-Level GPGPU Programming,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 78–90, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2010.62

[86] J. Docampo, S. Ramos, G. Taboada, R. Exposito, J. Tourino, and R. Doallo, “Evaluation of Java for general purpose GPU computing,” in *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, March 2013, pp. 1398–1404.

[87] P. W. Trinder, E. Barry Jr., M. K. Davis, K. Hammond, S. B. Junaidu, U. Klusik, H.-W. Loidl, and S. L. Peyton-Jones, “Low level architecture-independence of Glasgow parallel Haskell (GpH),” in *Glasgow Workshop on Functional Programming*, Pitlochry, Scotland, Sep. 1998. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/gph/papers/abstracts/low-level-gph.html

[88] S. Peyton Jones, A. Gordon, and S. Finne, “Concurrent Haskell,” in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, ser. POPL '96. New York, NY, USA: ACM, 1996, pp. 295–308. [Online]. Available: http://doi.acm.org/10.1145/237721.237794

[89] T. Harris, S. Marlow, and S. P. Jones, "Haskell on a shared-memory multiprocessor," in *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '05. New York, NY, USA: ACM, 2005, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/1088348.1088354

[90] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 325–335. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168898

[91] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for GPUs: (via language support for architectures and compilers)," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254066

[92] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL - A Portable Skeleton Library for High-Level GPU Programming," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1176–1182. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.269

[93] ——, "Towards high-level programming of multi-gpu systems using the skelcl library," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1858–1865. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2012.229

[94] A. Stromme, R. Carlson, and T. Newhall, "Chestnut: a GPU programming language for non-experts," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, 2012, pp. 156–167. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141720

[95] J. Hogg, "Islands: Aliasing protection in object-oriented languages," in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '91. New York, NY, USA: ACM, 1991, pp. 271–285. [Online]. Available: http://doi.acm.org/10.1145/117954.117975

[96] Mozilla, "The Rust Reference Manual," http://static.rust-lang.org/doc/0.9/rust.html, 2014, [Online; Accessed 17th Feburary 2014].

[97] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '98. New York, NY, USA: ACM, 1998, pp. 48–64. [Online]. Available: http://doi.acm.org/10.1145/286936.286947

[98] P. Haller and M. Odersky, "Capabilities for uniqueness and borrowing," in *Proceedings of the 24th European Conference on Object-oriented Programming*, ser. ECOOP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 354–378. [Online]. Available: http://dl.acm.org/citation.cfm?id=1883978.1884002

[99] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff, "A type system for borrowing permissions," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 557–570. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103722

[100] W. R. Stevens, *UNIX Network Programming: Networking APIs: Sockets and XTI*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[101] C. Doxsey, *An Introduction to Programming in Go*. CreateSpace Independent Publishing Platform, September 2012.

[102] J. Armstrong, "Making reliable distributed systems in the presence of errors," Ph.D. dissertation, Royal Institute of Technology, Stockholm, 2003.

[103] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/800028.808479

[104] B. G. Ryder, "Constructing the call graph of a program," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 216–226, May 1979. [Online]. Available: http://dx.doi.org/10.1109/TSE.1979.234183

[105] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy, "Constructing the procedure call multigraph," *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, pp. 483–487, Apr. 1990. [Online]. Available: http://dx.doi.org/10.1109/32.54302

[106] E. M. Myers, "A precise inter-procedural data flow algorithm," in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*,

ser. POPL '81. New York, NY, USA: ACM, 1981, pp. 219–230. [Online]. Available: http://doi.acm.org/10.1145/567532.567556

[107] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1981, ch. 7, pp. 189–234.

[108] ANSA, "ANSA: An engineer's introduction to the architecture," Architecture Projects Managment Limited, Poseidon House, Castle Park, CAMBRIDGE, CB3 0RD, UK, Tech. Rep., 1989.

[109] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 109–133, Feb. 1988. [Online]. Available: http://doi.acm.org/10.1145/35037.42182

[110] A. Hasler, I. Talzi, J. Beutel, C. Tschudin, and S. Gruber, "Wireless sensor networks in permafrost research-concept, requirements, implementation and challenges," *Proc. 9th Intl Conf. on Permafrost (NICOP 2008)*, vol. 1, pp. 669–674, 2008.

[111] Moteiv, *Tmote Sky Datasheet http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf*, 2006.

[112] P. Harvey, "Inceos: The insense-specific operating system," University of Glasgow, Tech. Rep., 2010. [Online]. Available: http://paul-harvey.org/papers/MSci_project_0501942.pdf

[113] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013.

[114] D. N. Antonioli and M. Pilz, "Analysis of the Java class file format," Tech. Rep., 1998.

[115] W. Pugh, "Compressing Java class files," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: ACM, 1999, pp. 247–258. [Online]. Available: http://doi.acm.org/10.1145/301618.301676

[116] R. Rivest, "The MD5 message-digest algorithm," United States, 1992.

[117] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[118] O. Sharma, J. Lewis, A. Miller, A. Dearle, D. Balasubramaniam, R. Morrison, and J. Sventek, "Towards verifying correctness of wireless sensor network applications using Insense and Spin," in *Proceedings of the 16th International SPIN Workshop on*

Model Checking Software. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 223–240, expereince using spin to verify the channel mechanism of the Contiki Insense. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02652-2_19

[119] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 4, pp. 2:1–2:36, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1328195.1328197

[120] D. F. Bacon and V. T. Rajan, "Concurrent cycle collection in reference counted systems," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 207–235. [Online]. Available: http://dl.acm.org/citation.cfm?id=646158.680003

[121] B. Williamson, *Developing IP Multicast Networks*. Cisco Press, 1999.

[122] J. N. Al-Karaki and A. E. Kamal, "Routing techniques in wireless sensor networks: A survey," *Wireless Commun.*, vol. 11, no. 6, pp. 6–28, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1109/MWC.2004.1368893

[123] J. Dedecker, "Ambient-oriented programming in AmbientTalk: combining mobile hardware with simplicity and expressiveness," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 196–197. [Online]. Available: http://doi.acm.org/10.1145/1094855.1094932

[124] N. Chechina, P. Trinder, A. Ghaffari, R. Green, K. Lundin, and R. Virding, "The design of scalable distributed Erlang," in *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Oxford, UK, 2012.

[125] W. R. Stevens, *TCP/IP Illustrated (Vol. 1): The Protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993.

[126] S. Farahani, *ZigBee Wireless Networks and Transceivers*. Newton, MA, USA: Newnes, 2008.

[127] A. Hasler, I. Talzi, J. Beutel, C. Tschudin, and S. Gruber, "Wireless sensor networks in permafrost research concept, requirements, implementation and challenges," in *Proc. 9th International Conference on Permafrost (NICOP)*, Jun 2008.

[128] J. Burrell, T. Brooke, and R. Beckwith, "Vineyard computing: Sensor networks in agricultural production," *IEEE Pervasive Computing*, vol. 3, pp. 38–45, January 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1435710.1437549

[129] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462. [Online]. Available: http://dx.doi.org/10.1109/LCN.2004.38

[130] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer, "On the need for system-level support for ad hoc and sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. 2, pp. 1–5, 2002.

[131] R. Mueller, G. Alonso, and D. Kossmann, "SwissQM: next generation data processing in sensor networks," in *3rd Biennial Conference on Innovative Data Systems Research*, 2007, pp. 1–9.

[132] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan, "Tosthreads: thread-safe and non-invasive preemption in TinyOS," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. New York, NY, USA: ACM, 2009, pp. 127–140. [Online]. Available: http://doi.acm.org/10.1145/1644038.1644052

[133] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182811

[134] E. Rondini and S. Hailes, "Distributed computation in wireless ad hoc grids with bandwidth control," in *Proceedings of the 5th international conference on Embedded networked sensor systems*, ser. SenSys '07. New York, NY, USA: ACM, 2007, pp. 437–438. [Online]. Available: http://doi.acm.org/10.1145/1322263.1322334

[135] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, "COOJA/MSPSim: interoperability testing for wireless sensor networks," in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, ser. Simutools '09. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, pp. 27:1–27:7. [Online]. Available: http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5637

[136] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[137] J. Fitzpatrick, “More C++ gems,” R. C. Martin, Ed. New York, NY, USA: Cambridge University Press, 2000, ch. Applying the ABC Metric to C, C++, and Java, pp. 245–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=331120.331161

[138] T. Hart and D. Edwards, “The alpha-beta heuristic,” Cambridge, MA, USA, Tech. Rep., 1963.

[139] R. Milner, *The Space and Motion of Communicating Agents*, 1st ed. New York, NY, USA: Cambridge University Press, 2009.

[140] L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, and H. Niss, “Bigraphical models of context-aware systems.” in *FoSSaCS*, ser. Lecture Notes in Computer Science, L. Aceto and A. Inglfsdttir, Eds., vol. 3921. Springer, 2006, pp. 187–201. [Online]. Available: http://dblp.uni-trier.de/db/conf/fossacs/fossacs2006.html#BirkedalDEHN06

[141] M. Calder, A. Koliousis, M. Sevegnani, and J. Sventek, “Real-time verification of wireless home networks using bigraphs with sharing,” *Science of Computer Programming*, vol. 80, Part B, no. 0, pp. 288 – 310, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642313001974

[142] M. Sevegnani and E. Pereira, “Towards a bigraphical encoding of actors,” June 2014. [Online]. Available: http://eprints.gla.ac.uk/94772/

[143] S. Gay, V. T. Vasconcelos, and A. Ravara, “Session types for inter-process communication,” School of Computing Science, University of Glasgow, Tech. Rep., 2003.

[144] N. Ng, N. Yoshida, X. Y. Niu, and K. H. Tsoi, “Session types: Towards safe and fast reconfigurable programming,” *SIGARCH Comput. Archit. News*, vol. 40, no. 5, pp. 22–27, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2460216.2460221

[145] R. Hu, N. Yoshida, and K. Honda, “Session-based distributed programming in Java,” in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 516–541. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70592-5_22

[146] J. Ellul and K. Martinez, “Run-time compilation of bytecode in wireless sensor networks,” in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ser. IPSN ’10. New York, NY, USA: ACM, 2010, pp. 422–423. [Online]. Available: http://doi.acm.org/10.1145/1791212.1791286

[147] R. Rozumalski, *WRF Environmental Modeling System - Users Guide*, release 2.1.2.2 ed., National Weather Service SOO Science and Training Resource Center, May 2006. [Online]. Available: http://archipelago.uma.pt/pdf_library/Rozumalski_2006_SOO\&STRC.pdf

[148] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.

[149] T. Ishiyama, K. Nitadori, and J. Makino, "4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 5:1–5:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389003

[150] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 232–245. [Online]. Available: http://doi.acm.org/10.1145/158511.158639