Actor-Oriented Programming for Resource Constrained Multiprocessor Networks on Chip

Gary J. Plumbridge

PhD

University of York
Computer Science

August 2015

for Ellie

Abstract

Multiprocessor Networks on Chip (MPNoCs) are an attractive architecture for integrated circuits as they can benefit from the improved performance of ever smaller transistors but are not severely constrained by the poor performance of global on-chip wires. As the number of processors increases it becomes ever more expensive to provide coherent shared memory but this is a foundational assumption of thread-level parallelism. Threaded models of concurrency cannot efficiently address architectures where shared memory is not coherent or does not exist.

In this thesis an extended actor oriented programming model is proposed to enable the design of complex and general purpose software for highly parallel and decentralised multiprocessor architectures. This model requires the encapsulation of an execution context and state into isolated *Machines* which may only initiate communication with one another via explicitly named channels. An emphasis on message passing and strong isolation of computation encourages application structures that are congruent with the nature of non-shared memory multiprocessors, and the model also avoids creating dependences on specific hardware topologies.

A realisation of the model called Machine Java is presented to demonstrate the applicability of the model to a general purpose programming language. Applications designed with this framework are shown to be capable of scaling to large numbers of processors and remain independent of the hardware targets. Through the use of an efficient compilation technique, Machine Java is demonstrated to be portable across several architectures and viable even in the highly constrained context of an FPGA hosted MPNoC.

Contents

Abstract 3

List of figures 9

List of tables 13

Acknowledgements 17

Declaration 19

1 Introduction 21

1.1 The Path to Networks-on-Chip 22

1.2 Multiprocessor Shared Memory 24

1.3 Challenges for Programming MPNoCs 25

1.4 The Actor Model 26

1.5 Thesis Hypothesis and Research Objectives 28

1.5.1 Hypothesis 29

1.5.2 Research Objectives 29

1.5.3 Thesis Contributions 30

1.6 Thesis Structure 30

2 Background 33

2.1 Multiprocessor Networks on Chip 33

2.2 Embedded System Design 35

2.3 Programming MPNoCs 38

2.3.1 Hardware-Specific Approaches 40

2.3.2 General Purpose Communication-Centric Programming 44

2.3.3 Summary 46

2.4 Actor-Oriented Design and Programming 46
2.4.1 Actor-Oriented Frameworks 48
2.5 Java 51
2.5.1 Java in Resource Constrained Contexts 52
2.5.2 Java in Hardware 53
2.6 Summary 54

3 Machine Abstract Architecture **57**
3.1 Machine Abstract Architecture 60
3.1.1 Systems 60
3.2 Application Model 61
3.2.1 Machines 67
3.2.2 Timing, Scheduling and Synchronisation 70
3.2.3 Intramachine Timing and Scheduling 71
3.2.4 Creating Machines 75
3.2.5 Communications 76
3.2.6 Dynamic Synthesis and Communication 95
3.2.7 Resource Usage 107
3.3 Platform Model 107
3.3.1 Processors 110
3.3.2 Resources 112
3.3.3 Communications Resources 114
3.3.4 Making Use of the Platform 116
3.3.5 A Platform API 117
3.3.6 Realisation of the Platform API 118
3.4 Frameworks and Runtime Behaviour 119
3.4.1 Processor Managers 120
3.5 Realisation of the Application Model 121
3.5.1 Requirements for Programming Languages 122
3.6 Summary 125

4 Machine Java **127**
4.1 Java 129
4.2 Machine Java 130

4.3 Machine Oriented Programming 131
4.3.1 Single-Thread Equivalence 134
4.3.2 Machine Classes 135
4.3.3 Machine Class Restrictions 140
4.3.4 Machine Life Cycle 144
4.3.5 Machine End-of-Life 147
4.4 Platform Agnostic Framework 150
4.4.1 Communications Channels 150
4.4.2 Implementing Channel Protocols 160
4.4.3 Spontaneous Event Sources 168
4.4.4 Processor Managers 169
4.5 Platform API 174
4.5.1 Platform Classes 176
4.5.2 Processors 178
4.5.3 Resources 179
4.5.4 Communications 179
4.5.5 The XYNetwork Platform 180
4.6 Implementation in Standard Java 184
4.6.1 Starting the Machine Java Framework 184
4.6.2 Machine Instance Identification 186
4.6.3 Machine Creation and Channel Addressing 186
4.6.4 Machine Isolation and Communication 189
4.7 Implementation on Bare-Metal Hardware 190
4.7.1 Compilation Strategy 193
4.7.2 Language Runtime 204
4.7.3 Network Runtime 213
4.8 Summary 221

5 Evaluation 223

5.1 Overview 224
5.2 Hardware Evaluation Platform 226
5.2.1 The Blueshell NoC Generator 226
5.2.2 Evaluation Platform Specification 230
5.3 Programming with Machine Java 232

5.3.1 Defining, Extracting and Fixing Application Structure 233
5.3.2 Application Platform Independence 246
5.3.3 Complex and Dynamic Application Structures 249
5.3.4 Concurrency Without Sharing 254
5.3.5 Fault Tolerance in Machine Java 262
5.4 Overheads and Scaling 265
5.4.1 Static Memory Consumption 265
5.4.2 Experimental Methodology 276
5.4.3 Communications Overheads 284
5.4.4 Computation Performance 298
5.5 Summary 306

6 Conclusion 309
6.1 Revisiting the Hypothesis 309
6.2 Thesis Contributions 311
6.2.1 Machine-Oriented Programming Model 311
6.2.2 Machine Java Framework 312
6.2.3 Chi Optimising Java Compiler 313
6.2.4 An Empirical Evaluation of Machine-Oriented Programming . . . 313
6.3 Future Work 314
6.3.1 Exploitation of Extracted Application and Platform Information . 314
6.3.2 Garbage Collecting Machines 314
6.3.3 Relocating Active Machines 315
6.3.4 Communications Performance Improvements 316
6.3.5 Multiple Applications 317
6.4 Epilogue 318

A Machine Java API Class Examples 319
A.1 RemoteProcedureCall.java 320
A.2 Period.java 325

B Water Tank Level Control Application 327
B.1 FlowController 327
B.2 RefillPump 328
B.3 LevelSensor 329

B.4 EmergencyValve . . . 331
B.5 UnreliableSensor . . . 332

C Microbenchmark Source Code **333**
C.1 SpeedTest . . . 333
C.1.1 SpeedTest.java . . . 333
C.1.2 PingClient.java . . . 337
C.1.3 PingServer.java . . . 339
C.1.4 PingClientReport.java . . . 340
C.2 SpeedTestHP . . . 341
C.2.1 PingClientHP.java . . . 341
C.2.2 PingServerHP.java . . . 343
C.3 DistributedMD5 . . . 344
C.3.1 DistributedMD5.java . . . 344
C.3.2 MD5Worker.java . . . 347
C.4 Dining Philosophers . . . 349
C.4.1 Fork.java . . . 349
C.4.2 Dining.java . . . 351
C.4.3 Philosopher.java . . . 352

D Sample Experimental Log **359**

E Scoped Memory Allocation in Chi **363**

F Glossary **369**

Bibliography **371**

List of Figures

3.1 The Machine Abstract Architecture 61
3.2 A water tank level control application 63
3.3 Extracted properties from example application 65
3.4 Extracted static structure in example application 65
3.5 Application model entity-relationship diagram 66
3.6 The logical local area of the Flow Controller machine. 68
3.7 Encapsulation in the example application 68
3.8 Gas cylinder machine 69
3.9 Scheduling of machine event handlers 73
3.10 “Can Send?” race condition 83
3.11 Event-driven non-blocking send 86
3.12 Non-blocking rendezvous construction 93
3.13 Platform model entity-relationship diagram 108
3.14 A detailed example platform 109
3.15 Logical connectivity in the example platform 111
3.16 Physical connectivity in the example platform 111
3.17 The **ProcessorManager** definition 120

4.1 The Machine Java stack 128
4.2 Hello world in Machine Java 134
4.3 A basic two-machine application 136
4.4 The functionality provided by the `Machine` class 138
4.5 An invalid static method definition 141
4.6 Machine validation procedure 143
4.7 The SetupableMachine variant 147
4.8 TPIF library structure 151

4.9 The EventSource abstract class 153
4.10 Immutable data validation procedure 159
4.11 **TPIFDriverTx** class contract 162
4.12 **TPIFDriverRx** class contract 163
4.13 **AlarmDriver** class contract 168
4.14 Spontaneous event source library 169
4.15 **ProcessorManager** class in Machine Java 170
4.16 newMachine() sequence diagram 171
4.17 The Platform abstract class 175
4.18 The Processor abstract class 178
4.19 A linear machine pipeline 183
4.20 A folded machine pipeline 183
4.21 The Machine Java compilation workflow 194
4.22 The Network-Chi Application Exploration Procedure 197
4.23 The Network-Chi Itinerary Construction Procedure 201
4.24 JVM Bytecode and Itinerary Examples 203
4.25 Application Throughput vs. Allocator Block Size 212
4.26 A minimal Network-Chi Java application. 215
4.27 A basic `Flattenable` class 217

5.1 A Bluetree memory network 228
5.2 The Evaluation NoC's MicroBlaze Tile 229
5.3 The Evaluation Network-on-Chip 230
5.4 Dependent Machines in Flow Controller Application 235
5.5 The internal architecture of an SDFCombiner 244
5.6 The `EventDivider` 245
5.7 An unsynchronised application pipeline 250
5.8 The `EventCombiner` 251
5.9 A synchronised application pipeline 252
5.10 The EventCombiners used by **LevelSensor** 253
5.11 Five dining philosophers 254
5.12 A Machine Java Fork 255
5.13 Initialising a static dining philosophers 256
5.14 Dynamic dining philosophers elaboration 257

5.15 Dynamic dining philosophers structure 257
5.16 Build options vs. static memory consumption 271
5.17 Code size survey 274
5.18 Code size survey 275
5.19 SpeedTest microbenchmark machine dependency graph 282
5.20 DistributedMD5 microbenchmark machine dependency graph 283
5.21 Serialisation of an application message 286
5.22 Message throughput on Blueshell 287
5.23 Message throughput on a PC 288
5.24 Processor count vs Blueshell message throughput 289
5.25 Processor count vs OS target message throughput 290
5.26 *Position-relative* allocation of SpeedTest machines 292
5.27 *Sequential* allocation of SpeedTest machines 293
5.28 Impact of processor iterator on message throughput 294
5.29 Complete throughput comparison of processor vs. machine count 295
5.30 Impact of channel protocol on message throughput 296
5.31 Communications throughput for the **Signal**-based SpeedTest 296
5.32 Computation throughput vs. machine count on Blueshell 299
5.33 Computation throughput vs. machine count on OS 299
5.34 Computation throughput with a minimal kernel 302
5.35 Computation throughput vs. batch size 303

List of Tables

3.1 Modelled unidirectional channels 91
3.2 Modelled bidirectional channels 91
3.3 Comparison of application elements 96
3.4 Interpretation of capability patterns 106
3.5 Implementation Language Comparison 124

4.1 Machine Java Representation of Application Model Entities 132
4.2 Destructive vs. non-destructive channels 154
4.3 Post-Mortem Heap Characteristics 211
4.4 Network-Chi's `Network` API 215

5.1 Common Bluetiles services 228
5.2 Sizes of Partial Application Compilation 236
5.3 Contributors to static memory consumption 267

E.1 A Scope-Based Memory API 365

Acknowledgements

I would first like to acknowledge the support of my supervisor, Neil Audsley, who attempted with some success to keep my head out of the clouds.[1] Neil's practical advice kept the process on track –over several bumps– and led me to an acceptable thesis. I'd also like to thank my two assessors, Leandro Indrusiak and Richard Jones for their bewilderingly thorough comments and a viva I'm sure I won't ever forget.

During my PhD I was lucky enough to work alongside Ian Gray, Jack Whitham and Jamie Garside. Their unreasonably extensive technical expertise and contrasting attitudes were so important to my work and development as a researcher and an engineer. I'd also like to extend my gratitude to David George and James Williams for their friendship and perspective over what has become many years. I also greatly appreciate Anthony Hatswell for adding more than enough motivation to complete what I started.

I'd like to thank my loving[2] wife, Sarah, for her support and encouragement through what has been a strange and trying experience. Finally, Eleanor Plumbridge (b. 1/10/2014) has been a challenge and a delight in equal measures, she really added a new texture to completing a PhD and I can't imagine it any other way.

[1] Although clouds are somewhat more fashionable in 2016.
[2] Of course, she would never admit it.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2009 - 2015. This work has not been previously presented for an award at this or any other university. Except where stated, all of the work contained within this thesis represents the original contribution of the author.

Parts of this thesis have previously been published in the following papers:

- Gary Plumbridge and Neil Audsley. Extending Java for Heterogeneous Embedded System Description. In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–6. IEEE, jun 2011. ISBN 978-1-4577-0640-0. doi: 10.1109/ReCoSoC.2011.5981527

- Gary Plumbridge and Neil Audsley. Translating Java for Resource Constrained Embedded Systems. *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, jul 2012. doi: 10.1109/ReCoSoC.2012.6322868

- Gary Plumbridge and Neil C. Audsley. Programming FPGA based NoCs with Java. *2013 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2013*, 2013. doi: 10.1109/ReConFig.2013.6732323

- Gary Plumbridge, Jack Whitham, and Neil Audsley. Blueshell: a platform for rapid prototyping of multiprocessor NoCs and accelerators. *ACM SIGARCH Computer Architecture News*, 41(5):107–117, 2014

  This work was the result of a collaboration and this is noted clearly in section 5.2.1.

Chapter 1

Introduction

Highly parallel multiprocessor network on chip (MPNoC) architectures are an active area of research and a great variety of speculative, prototype and commercial examples have been demonstrated. The MPNoC design pattern promises to better exploit the improvements in silicon manufacturing technology by reducing dependence on global on-chip wires. Long on-chip wires have not benefited so substantially from technological improvements and now represent an architectural bottleneck. The increased number of processors in MPNoC architectures makes it far more expensive to provide coherent shared memory, which creates a substantial problem: Without coherent shared memory familiar thread-based models of concurrency cannot be efficiently implemented. Other programming models that are not beholden to shared memory have been applied successfully in other domains, and this thesis proposes that such a model can be adapted to address the specific challenges of multiprocessor on-chip networks.

In this chapter the motivation for the work in this thesis is presented, briefly covering the following topics:

- The motivation for on-chip networks (section 1.1)
- The difficulty of scaling shared memory to large numbers of processors (section 1.2).
- The specific challenges for programming multiprocessor networks on chip (section 1.3).
- The actor model of concurrency, and its potential for MPNoC programming (section 1.4).

- Finally, the research hypothesis of this work is described in section 1.5.1.

1.1 The Path to Networks-on-Chip

Improvements in the manufacture of silicon integrated circuits (ICs) have led to a near continual reduction in possible transistor sizes. The two major consequences of this are that more transistors may be placed within a fixed area, and smaller transistors enable the maximum switching frequency to be increased. When the operational frequency of an IC is increased the power consumption also increases, but until nanometre scales power consumption is not affected by the transistor density of an IC. The scale-invariance of MOSFET[1] power density (identified by Dennard et al. in 1974 [53]) implies that increasing the transistor count in a fixed area of silicon can provide effectively 'free' additional capabilities. Increasing the transistor count of ICs has therefore become the preferred route to achieve higher performance as all computer systems are constrained in some way by power. Small embedded systems are typically constrained by the availability of power from low-capacity batteries or the local environment, whereas warehouse-scale High Performance Computing (HPC) is more often constrained by the costs of electricity and dissipating waste heat.

However, while smaller transistors enable higher transistor densities and therefore greater integration of complex functional units, Dennard et al. also identified that on-chip wires do not benefit significantly from a reduction in feature size. This means that as feature sizes decrease, the wiring delays become an ever greater performance bottleneck [204]; the RC[2] timing constants for on-chip wires remains approximately constant as they are scaled to smaller sizes so they effectively perform much worse than the scaled transistors which can switch substantially faster. The point at which global on-chip wires would become unfeasible had long been predicted [199, 32] and this point can be considered to have arrived soon after 2004 [60]. Coincidentally, this is similar to when Dennard's MOSFET scaling is considered to have ended [31], as around this time the power density of smaller transistors began to increase due to additional junction leakage and other effects of very small feature sizes.

For any given feature size, the absolute signalling delay of an on-chip wire is primar-

[1] *Metal Oxide Semiconductor Field Effect Transistor*

[2] Where R and C are the resistance and capacitance of the wires, respectively. R increases as wires are scaled down but C decreases by an approximately equal factor [204].

ily a function of its length. This implies that IC designs that can minimise wire lengths are better able to exploit the improved performance of scaled transistors. Historically, on-chip bus-type interconnects (*shared-medium* type interconnects in general) have been favoured for small numbers of processors and peripherals due to their simplicity and efficiency but these architectures are dependent on long wires; all devices connected to a bus share the same set of wires for communication. Long wires can be *pipelined* to enable high frequency operation and improve throughput [26, 157] but this increases the required silicon area and does not provide any improvement for signalling latency which is at best linearly related to interconnect length. Additionally, shared-medium interconnects scale poorly with respect to their number of *masters* (connected devices that can initiate transactions) as arbitration mechanisms are required to prevent simultaneous access by masters.

The problem of shared-medium interconnect scaling is particularly acute as the combination of abundant high performance transistors and low performance long wires also motivates many smaller on-chip processors rather than a single large processor. This argument for single-chip multiprocessors is now decades old [147], and is further supported by the observation that performance gains achieved via the exploitation of *instruction level parallelism* (ILP) have been exhausted [85] and therefore the explicit parallelism of multiple processors is required to secure further performance improvements.

Architectures where devices are organised into on-chip *networks* with many shorter point-to-point links are able to address the challenge of poor interconnect scaling as the maximum required wire length is reduced. These *Network-on-Chip* architectures can accommodate far greater numbers of on-chip devices (such as processors) as each network *node* only has connections to its neighbour nodes. This means that the complexity to add a new on-chip device is determined only by the network topology, i.e. how many neighbour nodes it will have, and not by the absolute number of nodes in the system. Increasing the number of nodes in a network also has the desirable consequence that the overall network bandwidth is also increased as additional routers and connections will also be added.

1.2 Multiprocessor Shared Memory

While Networks-on-Chip are a promising methodology for extremely large scale integrated circuits, such as single-chip multiprocessors with thousands of discrete processing units, the orthogonal problem of scaling access to shared memory for the processing units is not addressed by these *communication-centric* architectures. This represents one of the many challenges of programming multiprocessor networks-on-chip (MPNoCs). *Symmetric* shared memory is a significant bottleneck for multiprocessor systems as memory bandwidth is shared between all processors and therefore aggregated demand for memory bandwidth can easily exceed the capabilities of the memory device.

Processor-local caching of shared memory can substantially reduce the overall demand for memory bandwidth, enabling far larger numbers of processors to be supported for a given memory capability. However, the existence of multiple caches leads to the *cache coherence problem* [80, §5.2], where caches may have differing views of the same regions of shared memory. The *coherence* of caches is enforced by the implementation of cache coherence protocols. Simple *snooping* coherence protocols that are effective for small numbers of processors quickly become intractable as the number of caches increases, but more sophisticated techniques such as *directory-based* protocols are complex and remain difficult to scale [4, 48]. It is not universally accepted that scaling cache coherence to large numbers of processors is impossible [129], at least for the near future. However, the cost of coordinating caches is necessarily related to the total number of caches sharing the same region of memory. No matter how marginal this cost is there will still come a point where the expense of adding additional caches is dominated not by the cache itself but by its interaction with the other caches.

Aside from the challenges of presenting a coherent view of shared memory to each processor, programming using a multithreaded model of concurrency (which depends on the availability of shared memory) is very challenging: Processors must *synchronise* with each other to ensure that shared data structures are always consistent when accessed concurrently but this is a notoriously error prone endeavour [198]. Incorrect synchronisation of concurrent accesses to shared data can result in race conditions, livelocks and deadlocks. For these reasons and in spite of its ubiquity, it has been suggested that threaded concurrent programming models be abandoned [116] in favour of more explicit models of concurrency – even on hardware architectures that can provide coherent shared memory.

If programming models that do not depend on shared memory are applied to multiprocessor networks on chip, then the hardware complexities due to implementing cache coherence protocols can also be avoided.

1.3 Challenges for Programming MPNoCs

Overall, it is far from straightforward to design software for multiprocessor networks-on-chip, especially those without coherent caches or without shared memory at all. The challenges include:

1. The familiar thread-parallel programming model available in many general purpose programming languages (including C [97], Java [74] and Ada [71]) cannot be supported efficiently without shared memory. Languages that only support concurrency via threads cannot effectively specify the behaviour of a non-shared memory multiprocessor system [75].

2. Coordination of processor activity via network communication is essential to exploit an MPNoC's capabilities. This creates challenges for programming language design as communications capabilities must be exposed to an application without introducing hardware-specific dependencies unnecessarily.

3. Networks-on-Chip can have irregular topologies and heterogeneous functional units. Additionally, non-functional and dynamic constraints (such as power, temperature or voltage) can be viewed as a type of implicit heterogeneity. Heterogeneity further complicates the challenge of efficiently exploiting the hardware while also limiting the hardware-specificity and complexity of the software.

4. Network-on-Chip architectures do not necessarily require global clock synchronisation; each network node can reside in its own clock domain and the network interconnect can hide the complexity of communication between clock domains [145]. This means that there is not necessarily a consistent interpretation of time between any two processors, and this complicates temporal synchronisation of processors.

5. Minimisation of on-chip wire lengths provides an incentive to reduce the size and complexity of the synchronous circuits in each clock domain, meaning that there

may be an advantage to using more simple processors over fewer sophisticated processors. Simple processors will have more limited capabilities and smaller local memories, and this presents a challenge for programming as it limits the selection of suitable programming languages and libraries.

While it is possible to design software for each processor in an MPNoC individually it is much more desirable to have a software methodology that enables a whole network of processors to be programmed with a single software application.

1.4 The Actor Model

Although threaded-models of concurrency are very poorly suited to MPNoC architectures, *actor-oriented* models of concurrency are particularly well suited to the challenges. *Actors* were first described by Hewitt in 1973 [81] as a theoretical universal model of computation, but were later refined by Agha in 1985 [5] as a suitable model of concurrent computation for distributed systems.

Fundamentally, the actor model of computation considers a system to be a collection of actors that can *asynchronously* send messages to one another. When an actor receives a message it can do any of the following concurrently:

1. Send messages to other actors.
2. Create new actors.
3. Redefine its behaviour for subsequent messages received.

Actors are considered to have *unforgeable* 'addresses' – they cannot be guessed or enumerated by an actor. An actor can only send messages to actors that it already has the address of, and addresses can only be acquired by creating new actors or receiving a message containing one or more addresses. Actors themselves do not have explicit internal *state* other than the behaviour defined for the next message received. However, the expression that defines the behaviour of the actor has a list of *acquaintances* (addresses of other actors) that is bound when the behaviour is assigned to the actor. Values, such as integers, are represented by so-called *pure* actors that have their behaviour bound with acquaintances that are never changed; the behaviour of a pure actor is defined not to redefine its own behaviour after creation. Values, operators and processes are all

actors in this model. Although elementary, the actor model is universal and has been shown to be sufficient to express programming language control structures [23] and data structures [5]. Even at the earliest stage in the development of the actor model Hewitt explained that actors can be thought of as *virtual processors* that are never so busy that another message cannot be sent to them [81].

As a universal model of computation, the actor model is not practical for direct realisation but it can and has been used as the theoretical basis for a large variety of programming languages and frameworks in other contexts. These realisations notably include Erlang [104], Scala [77] (and also via the Akka [91] framework) and Ptolemy II [3]. When used as a theoretical foundation for languages or frameworks, the implementation language will already contain the ability to express control and data structures in a considerably more succinct and practical way than with an actor abstraction. The concurrency aspects of the actor model are what motivates its use, and the important concepts of actor-modelled concurrency are:

- Actors interact only by *message passing*.
- Message passing is asynchronous and message queues have unbounded length.
- Actors execute concurrently with respect to one another.
- Messages can be *arbitrarily* reordered.
- Communication is unidirectional: an actor cannot reply to a message unless the sender identified itself.

For each of the MPNoC programming challenges described previously it can be argued that an actor model is either an appropriate response to the challenge or irrelevant:

1. The allowable actions for an actor's behaviour does not include the ability to inspect another actor's behaviour (internal state). This implies that an absence of shared memory is not a problem.
2. Actors are also communication centric.
3. Actor-oriented applications would be no worse for heterogenous architectures than thread-modelled applications.

4. Actors pass messages asynchronously and have no other defined relationships. The absence of a global timing source is not an issue for an actor-oriented application.

5. The suitability of an actor-oriented application to execution on a resource constrained architecture is entirely dependent on the implementation methodology; implementation overheads are not a well defined concept at the model-level of abstraction.

1.5 Thesis Hypothesis and Research Objectives

This thesis proposes that an actor oriented model of concurrency is a promising route for programming whole MPNoC architectures with single software applications, and specifically without requiring the hardware to provide expensive and (eventually) unscalable cache coherence protocols.

Some relevant comments can be found in the literature concerning the wisdom of investigating models for concurrent programming, and for applying these programming models to distributed systems:

> *"The message is clear. We should not replace established languages. We should instead build on them. However, building on them using only libraries is not satisfactory. Libraries offer little structure, no enforcement of patterns, and few composable properties."*

– E. Lee, "The Problem with Threads" [116, §7]

> *"There are fundamental differences between the interactions of distributed objects and the interactions of non-distributed objects. Further, work in distributed object-oriented systems that is based on a model that ignores or denies these differences is doomed to failure, and could easily lead to an industry-wide rejection of the notion of distributed object-based systems."*

– Waldo et al., "A Note on Distributed Computing" [214, §1]

The message to extract from these comments is that *structure*, *pattern enforcement* and *composability* are important to a successful concurrent programming model, but this structure must effectively distinguish between activities that happen locally and (in the context of an MPNoC) those that might happen on a remote processor.

1.5.1 Hypothesis

This thesis addresses the following hypothesis:

> *Resource constrained multiprocessor networks on chip with non-coherent caches can be programmed effectively with a general purpose programming language through the application of an actor-oriented model of concurrency.*

For the purposes of this thesis *resource constrained* processors will be considered to have <100KiB of processor-local writable memory including any caches, and <1MiB of read-only memory. These figures align broadly with the capabilities of modern off-the-shelf microcontrollers [144, 195, 94], and are comparable to the memory available to each processor in Intel's SCC MPNoC [131]. These memory constraints are driven by the expectation of MPNoCs constructed from vast arrays of limited processors. Results applicable to such limited processors will likely be applicable to more capable processors.

An MPNoC will be considered to be 'effectively' programmed primarily if it is demonstrated to be feasible (validated by existence), but also if the programming methodology conveys at least some advantage, such as arguments for improved scalability, improved application analysis or reduced scope for programmer errors. The costs of an 'effective' approach, in terms of resource overheads and programmer effort, must not be fundamentally intolerable; prohibitively expensive overheads must not be an intrinsic property of the approach.

Finally, the application of the an actor oriented model of concurrency to a *general purpose* programming language avoids constraining the approach to an artificially narrow application domain. Specialisation can always yield improved results for any metric, but the feasibility of using a conventional general purpose language to design software for highly multiprocessor on-chip networks is what is interesting.

1.5.2 Research Objectives

To test the hypothesis, the research objectives of this thesis are to:

1. Identify previous work relevant to the programming of networks on chip and explore how its strengths can be built upon.

2. Define a programming model based around the actor-oriented model of concurrency that will enable the exploitation of massively multiprocessor networks on

chip without coherent caches.

3. Construct a reference implementation of the MPNoC-appropriate programming model in an existing general purpose programming language.

4. Evaluate the characteristics of the model via the reference implementation, and the non-model characteristics specific to the reference-implementation itself, on a representative hardware architecture. The discovered characteristics of the model will enable the evaluation of the research hypothesis.

5. Identify the open questions that remain at the end of this research.

1.5.3 Thesis Contributions

This thesis provides a number of significant contributions, including:

- A specialised actor-oriented programming model intended for communication-centric and resource constrained architectures.
- A reference implementation of the programming model in the Java programming language: *Machine Java*
- An effective procedure for compiling Machine Java applications for execution on resource constrained platforms.
- An empirical evaluation of *machine-oriented* programming, including the overheads implied by this approach.

A more detailed summary of these contributions may be found in section 6.2.

1.6 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 provides a brief overview of relevant literature necessary to contextualise the work presented, with a particular focus on programming multiprocessor systems and networks on chip, actor oriented programming, and the use of the Java programming language in resource constrained contexts.

Chapter 3 describes the *Machine Abstract Architecture* (MAA): a set of models for application software, multiprocessor communication-centric hardware, and the limited

connection between the two. The application model enables the description of *machine-oriented* applications in which the actors (*machines*) may have any number of typed communications *channels* with explicitly defined characteristics.

Chapter 4 presents the *Machine Java* framework: a complete reference implementation of the Machine Abstract Architecture in the Java programming language. Machine Java is a high-level machine-oriented programming framework that enables the Java programming language to effectively exploit resource-constrained MPNoC architectures. This chapter also presents a compilation methodology to enable execution of Java and Machine Java applications on highly limited embedded processors.

Chapter 5 considers and discusses the claims made for the MAA application model, and for Machine Java as a programming framework. In particular the viability of programming resource constrained platforms is considered, the likelihood that this programming model and implementation can scale to large MPNoCs, and the difficulty of expressing applications in such a framework.

Finally, 6 concludes this thesis with a summary of contributions and discussion of the extent to which the hypothesis can be considered satisfied. A brief discussion of promising avenues for future work is also provided.

Following the conclusion several appendices (page 318 onwards) provide reference material for some Machine Java classes and important application examples including the microbenchmarks. A bibliography can be found at the end of the document starting at page 371.

Chapter 2

Background

Contents

1.1 The Path to Networks-on-Chip . . . 22
1.2 Multiprocessor Shared Memory . . . 24
1.3 Challenges for Programming MPNoCs . . . 25
1.4 The Actor Model . . . 26
1.5 Thesis Hypothesis and Research Objectives . . . 28
1.5.1 Hypothesis . . . 29
1.5.2 Research Objectives . . . 29
1.5.3 Thesis Contributions . . . 30
1.6 Thesis Structure . . . 30

In the previous chapter the challenges of programming non-shared memory networks-on-chip were established, particularly when the processors have limited local resources. While there have been few attempts to address this domain specifically there is a wealth of relevant literature addressing related architectures and issues. In this chapter a selection of previous multiprocessor network- and system- on chip approaches are discussed in section 2.3, approaches to the application of the actor model to programming frameworks are discussed in 2.4, finally in section 2.5 the use of Java in resource constrained contexts is considered.

2.1 Multiprocessor Networks on Chip

Multiprocessor network on chip architectures were introduced in the previous chapter as a technique for moving beyond the limitations of bus[1]-based multiprocessor configu-

[1] Generally any *shared-medium* interconnect.

rations. This section aims to provide some context for MPNoC architectures.

When discussing MPNoC architectures the emphasis is on the networked nature of the interconnect and not on any particular processor architecture, but the interconnect between the processors and other functional units (in the case of MPSoCs) is less important to the purpose of the MPNoC than the type or instruction sets of the processors provided. In this sense NoC-based designs can be considered general purpose, or more accurately: *unspecialised*. To a first-order approximation the suitability of an MPNoC for a given task depends on the suitability of it's embedded processors for that task. For this reason MPNoCs have been applied to varied problem domains, including:

- Scientific workloads including astrophysics, geophysics, material sciences and scientific visualisation. [96]
- Video games: The Cell Broadband Engine[2] [44] embedded within Sony's PlayStation 3 game console is a double ring-network architecture with nine processors (the so-called *Power Processing Element* (PPE) and *Synergistic Processing Element* (SPE) units), a memory controller and two IO interfaces.
- Network and cellular infrastructure including base stations, gateways, routers and real-time data analysis of network traffic.
- Video and signal processing. [96]
- Network Function Virtualisation [47] (NFV) where network architectures (again composed of gateways, routers, firewalls, etc.) are virtualised rather than implemented in real hardware. [143]

The examples above draw primarily from the use cases for the commercially available Tilera Tile64 [221] and its successors, and Intel's general purpose Xeon Phi coprocessor [103, 96], also based on a NoC interconnect architecture. It can be seen from this list that although the applications are diverse they also tend to be *stream* oriented. The stream processing paradigm is particularly well suited to highly parallel hardware architectures, so it is no surprise to see MPNoC architectures being applied to these problems.

A principle motivation for multiprocessor architectures is power reduction; it can be more efficient to apply many less-capable cores to a task than to use a single extremely

[2]Most often referred to as the *Cell* or *Cell B.E.*

high performance core. However, on-chip networks can be particularly power intensive [171] and therefore may not be perfectly suited to low power applications such as mobile devices. High power consumption in an unoptimised network-on-chip design is caused by the large number of signal transitions within network routers and buffers as packets progress through the network. Work has been undertaken to address the power consumption of NoC interconnects by introducing techniques such as transition minimising coding [155] and reduced-swing voltage transitions [171].

2.2 Embedded System Design

With the exception of MPNoC architectures destined to be coprocessors in standard personal computers (such as the Xeon Phi [103]), all MPNoC instances will become part of an embedded system. For this reason a brief coverage of some embedded system design topics is justified.

The term *embedded system* is used to refer to an instance of a special purpose computer functionally contained within another system that is not necessarily identified as a computer itself. The name also implies a level of application-specificity as an embedded system is typically produced for a single purpose and presented as a non-divisible unit. Examples can be found in every walk of life, from mobile phones to industrial heavy machinery, and are far too numerous to exhaustively list but it is sufficient to say that they are ubiquitous.

Embedded systems, like all others, require explicit design and there are several different high-level approaches: The most striking design choice available to embedded systems engineers is just how much of the behaviour of the system needs to be implemented by software or configurations embedded into the device and how much can be provided by dedicated fixed function hardware. The process of designing a system with both software and application-specific hardware components is known as *co-design* and is a nebulous topic. Although some embedded systems may be constructed using only fixed-function hardware or general-purpose hardware with fully software-defined behaviour, these approaches would not be considered co-design.

If an embedded system is to be co-designed then the tasks it must accomplish need to be *partitioned* between hardware and software and this requires an appreciation of what dedicated hardware is suited to, and what is more suited to implementation in

software. Thomas et al. [205] go some way to characterising what may be particularly suited to implementation in hardware:

arbitrary arithmetic operations Hardware can be designed to implement arithmetic on arbitrarily wide operands and over complex operations that are "expensive or clumsy" [205] in software. Reversing the bit order in a byte is an example of an operation that is expensive to implement in software on a general purpose processor but exceedingly fast and cheaply (it can be implemented with only wires and no logic) on a machine with dedicated support for this operation.

data parallelism Hardware is well suited to tasks where operations can be performed on multiple items of the data-set simultaneously. This is fine-grained parallelism.

task parallelism is well suited to hardware as for each concurrent task a whole new hardware unit can be placed to allow for its parallel execution rather than time multiplexing as on a single processor. This is coarse-grained parallelism. Thomas et al. [205] note that "multiple threads of control" are particularly well suited to hardware implementation as only an extra state machine needs to be implemented for each extra thread of control on the same task.

custom memory architectures are a significant advantage of hardware implementations. Contention for system memory bandwidth can be avoided by providing hardware execution units with their own memories to store their data-sets or temporary information.

These observations are broadly supported in literature, with Wirth [225] placing specific emphasis on fine-grained parallelism noting that the strength of hardware lies in its provision for unlimited amounts of parallelism. Ward and Audsley [217] argue that hardware implementations can be more conducive to timing analysis and therefore possibly more useful in realtime systems than general purpose processors.

In contrast, software can be used for any task that hardware is not *particularly* excellent at. From the observations that hardware is excellent at highly parallel, arbitrarily sized and predictable tasks, it could be inferred that sequential, fixed-arithmetic and dynamic tasks will be suited to a software implementation. However, it would not be appropriate to say that software will be *better* at these tasks but possibly nearly as good as a dedicated hardware implementation. Asanovic et al. [11] highlight the trade-off between efficiency and productivity which is particularly applicable to systems co-design.

Systems that are designed at a low level (such as primarily hardware systems) are likely to perform very well whereas abstract, high-level systems (such as general purpose processors) will be less efficient but have the advantage of superior productivity from the engineers involved.

Common embedded systems methodologies involve first modelling the intended behaviour of the overall system in a highly abstract way and then working towards an implementable design by successively refining the model to lower levels of abstraction. To an extent, *Hardware Description Languages* (HDLs), such as VHDL [13, 55] or Verilog [49] can be used to model a complete system, but more sophisticated approaches such as high level synthesis and system-level modelling have since been developed.

HDLs are primarily characterised by enabling a designer to specify a system using a structural domain of abstraction; they allow systems to be described in a hierarchical style, where the whole system is represented as a tree of components containing subcomponents. In the case of both VHDL and Verilog a system can also be expressed behaviourally at the *Register Transfer Level* (RTL) level of abstraction, where the system is described in terms of how data is manipulated in transfers between specific registers or latches. Although this process is now mature and tools exist to automatically *synthesise* hardware from HDL specifications, the level abstraction is far too low to cope with the complexity of modern embedded systems.

In response to the work required to design a system in hardware description languages, various *High Level Languages* (HLL) have been developed to enable the specification of hardware using syntax and semantics that much more closely resemble those of conventional programming languages. When describing the system behaviour in an HLL the eventual target hardware (such as an FPGA) is almost completely abstracted by the compiler [216]; the programmer doesn't need to care about the functional units that will be generated as a result of their code, nor the functional units that the target device already contains. Handel-C [6] is a notable example of an HLL as its syntax is significantly similar to that of standard C [97] but its runtime semantics are markedly different: It had a rigid timing model and poor support for global memory and data pointers. A more recent effort in high level synthesis is Xilinx's Vivado HLS tool [230] although this also provides a C language subset [224]. HLL approaches, in spite of their name, are still quite limited in their ability to abstract a system's behaviour but this is because they are primarily intended for the design of limited system components such

as IP cores or function accelerators.

The ability to abstractly capture a system's characteristics is directly addressed by system-level modelling tools (or *system specification* languages). There are several tools for system-level modelling including National Instruments' Labview and Mathworks' Simulink but in the context of multiprocessor embedded systems, SystemC [156] and Ptolemy II (see section 2.4.1.1) are of particular interest. SystemC is a C++ class library to enable the description and simulation of systems at multiple levels of abstraction. The power of this approach stems from the use of the C++ language enabling object oriented and templated system descriptions. Although system-level modelling can allow elegant descriptions of both application behaviour and platform structure, the realisation of the application from a simulatable model onto the real hardware is a large and challenging topic: the modelled application must be partitioned and mapped to the platform. Neither application partitioning nor mapping are trivial activities, in some cases application mapping can become an NP-hard activity [206]. This is because all of the non-functional application constraints must also be satisfied, and constraint satisfaction is not a simple procedure. Application mapping techniques aware of several different characteristics have been investigated including energy [206], performance [87], timing [186] and network contention [126].

2.3 Programming MPNoCs

The challenges of developing multiprocessor *systems* on chip (MPSoC) have been well identified in the literature [127], including issues related to the design of hardware and software. The important hardware concerns include:

- Identification of the number and architectures of the processing units.
- Identification of the on-chip interconnects between the processors and peripherals.
- Identification of an appropriate memory architecture to support the processors.

[127] also identify the following software-related challenges:

- What programming model is appropriate, and how will concurrency and synchronisation be supported?
- How can an application be partitioned across the processors and what APIs are appropriate?

The issue of scalability is identified as a problem common to both the hardware and the software design of a system.

Networks on chip represent a potential solution to the scalable design of complex on-chip interconnects, but the existence of a network does not readily imply any particular programming model that can also address scalable software. Theoretical models of computation that can achieve certain guarantees (such as bandwidth guarantees) on NoC architectures [102] have been developed, but these are concerned only with the communications characteristics of the architecture and not how the models can be applied to programming abstractions for multiprocessor networks. This problem, that multiprocessor *networks* on chip are difficult to program, has been widely recognised in the literature [124, 109, 65, 211]. The selection of appropriate abstractions for programming such platforms is considered to be the primary issue, as the complexity of MPNoC architectures is overwhelming when the hardware elements (the processors, routers and interconnections) are approached separately. Effective abstractions are required to hide replicated and irrelevant hardware characteristics from application software.

Martin [127] provides a short but incomplete appraisal of the difficulties of programming multiprocessor systems:

> *"Two factors are key: concurrency, and "fear of concurrency"*

This assessment fails to recognise many other concepts that can be troublesome in the design of parallel software. Skillicorn and Talia [192] elaborate five such concepts that can be abstracted or exposed by programming models. These concepts are *concurrency, decomposition, mapping, communication, and synchronisation*, and are used to form six levels of abstraction that can be used to categorise parallel programming models:[3]

1. **Implicit concurrency**: All parallel programming concepts are abstracted. The Haskell functional programming language is typical of this model.
2. **Parallel level**: Decomposition, mapping, communication and synchronisation are abstracted.
3. **Thread level**: Mapping, communication and synchronisation are abstracted.
4. **Agent models**: Communication and synchronisation are abstracted.

[3] This topic is summarised more clearly in [166, §2.2.1] than Skillicon's [192] thorough survey.

5. **Process networks**: Only synchronisation is abstracted.

6. **Message passing**: No concepts are abstracted.

Skillicorn asserts [192] that the implicit concurrency model will result in the simplest and least complex applications, but the realisation of such abstract programming models depends on the sophistication of the tooling. As this hierarchy of parallel programming model abstractions is not concerned with implementation details it is itself quite abstract. The representation of communications and concurrency abstractions has a significant impact on the efficiency and ease of use for an application programming framework.

In the remainder of this section a selection of programming frameworks and models relevant to MPNoCs are considered.

2.3.1 Hardware-Specific Approaches

The relative novelty of multiprocessor networks on chip means that many programming approaches directly related to MPNoC architectures have focussed on addressing the challenges specific to one particular MPNoC architecture. Early work in this field concentrated on the generation of MPNoC architectures that could be implemented on FPGA[4]-based reconfigurable logic and also supplied a platform-specific programming model. These approaches included xENoC [106], HeMPS [39] and RAMPSoC [69, 70].

2.3.1.1 xENoC

An early approach for MPNoC programming was provided by the xENoC [106] NoC experimentation framework. xENoC distinguished itself from previous work by providing a clear programming model that is distinct from its hardware-software co-design efforts. Previous literature had only focussed on the derivation of NoC architectures from software models, or on the programming challenges of system-on-chip architectures.

The xENoC framework generates a synthesisable NoC implementation from a selection of routers with a NIOS-II [8] processor at each node, and provides a programming model based on a subset of *Message Passing Interface* [215] (MPI). MPI is a comprehensive and de-facto standardised programming interface for designing message-passing

[4] *Field-Programmable Gate Array* (FPGA)

distributed applications. However, due to the substantial size and complexity of MPI (over 100 functions in MPI-1 and substantially more in later versions), xENoC presented a greatly reduced subset called *eMPI* that was more appropriate for the constraints of their MPNoC architectures. eMPI only has six functions, four of which are for initialising and finalising their framework, and a function each for send and receive. Less than one month after the publication of the xENoC work, the embedded-capable *Multicore Communications API* [202] (MCAPI) message-passing interface was completed [84]. The MCAPI software API was designed specifically to accommodate embedded heterogeneous architectures that could not be addressed by MPI due to its complexity and high memory requirements.

2.3.1.2 HeMPS

Soon after xENoC, one of the simplest MPNoC-centric programming models was presented as a component of the HeMPS [39] MPNoC-generation framework. As with xENOC, this work was primarily focussed on enabling the design space exploration of MPNoC architectures but this time built from Hermes [137] routers with Plasma [181] MIPS processors. The HeMPS programming framework is based around a *Kahn Process Network* [107] (KPN) distributed model of computation. Kahn process networks model a system as a collection of sequential processes that are interconnected by unbounded FIFO (first-in-first-out) buffers. The structure of the network is application-defined but static and each KPN process is deterministic. The HeMPs programming framework allows applications to be expressed as a set of tasks in the C [97] programming language. A microkernel present on each of the plasma processors provides round-robin preemptive multitasking and communication between tasks via a simple API. Although the KPN-modelled application task structure is static, the microkernel enables a limited form of dynamism where tasks are only allocated to a specific processor on their first use. While the programming model is a fairly simple two-function API, network reads (*ReadPipe()*) are blocking operations and must specify which task they wish to receive data from before the data can be received. This restriction complicates general purpose programming.

2.3.1.3 RAMPSoC

Another MPI-based programming approach [70] was demonstrated for the RAMPSoC [69] *system* on chip generator. The RAMPSoC framework demonstrated the possibility of systems on chip that can alter their architecture (including changes to processors and communications infrastructure) dynamically according to runtime application requirements. The RAMPSoC programming API contains substantially more functionality than xENoC's eMPI framework, covering 18 MPI functions, but required over 40KiB of program code. MPI-subset based programming models have proven very popular for systems- (and networks-) on chip architectures with many other demonstrated implementations [185, 125, 88, 223, 136, 173].

2.3.1.4 XC (XMOS)

Hardware-specific programming frameworks have also been demonstrated for non-reconfigurable network-on-chip architectures including Intel's *Single Chip Cloud* (SCC) and XMOS many-core chips. In the case of the XMOS many-core processors [133] an entirely new programming language is presented, XC [218], that provides an Occam [95]-like programming model in with C syntax. Occam's model of concurrency is a natural fit for highly parallel communication-centric architectures as it allows fine-grained concurrency to be expressed very easily: a `par` statement is provided that executes each of its containing statements concurrently. In the following example reproduced from [218, §3.2.1], lines 4 and 5 will execute concurrently:

```
1 int main(void) {
2   int i = 1, j = 2, k = 3;
3   par {
4     i = k + 1; //Thread X
5     j = k - 1; //Thread Y
6   }
7 }
```

The `par` block finishes after all of its spawned child threads have finished; it is a *fork-join* model of concurrency. Although the concurrent activities are referred to as threads in XC, data can only be shared between threads if it satisfies so-called 'disjointness rules':

- If a thread modifies a variable then no other thread can access that variable.
- If a thread accesses a variable then no other thread can modify it.

- If a thread uses a *port* (access to external hardware via IO pins) then no other thread can access that port.

These rules ensure that data cannot be shared in a way that would cause a race condition. As in Occam, communication is facilitated between threads with a channel concept rather than by sharing variables.

The same combination of Occam-like concurrency with C-syntax was also used successfully by Handel-C [6, 123], a language for high-level hardware synthesis.

2.3.1.5 Intel SCC

Intel's 48-core SCC [86] MPNoC exhibits both message-passing and non-coherent shared memory features in the hardware. Each of the 48 processors are complete 32-bit Intel architecture (P45C) processors with 256KiB of unified L2 instruction and data caches and have access to off-chip external memory via one of four on-chip DDR3 memory controllers. No cache coherence is provided by the hardware so applications must be especially careful if off-chip memory is to be used for process coordination [131], but an extension to the P45C's instruction set is provided to improve the efficiency in this case. The P45C processors have an extra *message passing buffer type* (MPBT) flag in their page tables to mark memory regions as belonging to data that will be shared between processors. Accesses to MPBT-flagged memory areas bypass the processor's L2 caches, and any data in L1 caches with an MPBT flag can be invalidated in a single clock cycle with a new instruction: `CL1INVMB`. This combination of hardware features enables more efficient software-driven cache coherence as only the areas of memory that can be modified by other processors need to be marked with the MPBT flag.

The sophistication of the SCC's processors enables them to execute full Linux kernels, and the presented programming platform [131] provides Linux network drivers to allow Linux applications to use TCP/IP for inter-core communications. A library-based communications framework (RCCE) is also presented to enable programming the SCC without an operating system. The RCCE library provides 'put' and 'get' primitives to move data to and from other processor's L1 caches. When used without an operating system each processor executes the same binary code and differentiates its behaviour based on a processor-specific ID provided by the RCCE library. This is an instance of the single-program multiple-data (SPMD) pattern.

Possibly the least accessible MPNoC programming methodology for a fabricated

chip was revealed by Intel's publication [132] concerning their experimental 80-processor 'Terascale' MPNoC: No compiler, framework or operating system is defined at all for this architecture; all applications must be hand assembled. However, the hardware itself is a true example of a resource constrained communication-centric MPNoC: each processor only has 3KiB of local instruction memory and 2KiB of local data memory, and there is no shared memory at all.

2.3.2 General Purpose Communication-Centric Programming

The abundance of MPI-type programming interfaces demonstrated for specific platforms strongly indicates the applicability of this model to communication centric architectures in general. A variety of platform-independent implementations of MPI are available including MPICH [197] and OpenMPI [64]. MPI is well established in high-performance computing [197] and bindings are available in many languages including C, C++, Fortran and Java. However the large required code sizes of MPICH and OpenMPI (47MiB and 40MiB, respectively [70]) excludes their use in even state of the art embedded systems. High capability platforms such as the SCC would be able to execute MPI applications, but a substantial performance penalty can be expected as the framework code is substantially larger than each processor's local caches; each SCC processor would repeatedly have to fetch communications framework instructions from off-chip RAM.

2.3.2.1 rMPI

rMPI [173] is a complete implementation of MPI requiring only 160KiB of code, enabling its use in embedded and general purpose multiprocessor system on chip architectures. rMPI was validated on the Raw [201] processor but even with its vastly reduced code footprint compared to MPICH or OpenMPI, it was still much larger than the local memory available to the processors it was demonstrated on. The Raw architecture is a 4×4 mesh network of MIPS-like processors each with 32KiB of hardware-managed data caches and 32KiB of *software-managed* instruction caches. Much like the SCC, Raw supports access to off-chip memory enabling larger applications to execute than can fit into processor-local memory. Application performance using rMPI on the Raw processor was shown to be highly sensitive to processor-local instruction cache size. The authors conclude that MPI is simply too large and complex to be well-suited for multiprocessor

systems on chip.

Although MPI is by far the best-represented approach for MPSoC programming, other interfaces and models have also been considered, including: Non-MPI message passing abstractions (NoCMSG [235]), tool-assisted parallelisation of existing legacy C code (MAPS [42] and IMEC's approach [135] using MPA [22]), and novel models specifically for communication-centric architectures (TinyGALS [46] and HyMR [158]).

2.3.2.2 TinyGALS

TinyGALS [46] provides a *globally-asynchronous locally-synchronous* (GALS) [43, 139] programming model for event-driven embedded systems such as wireless sensors networks (WSNs). Although this work was motivated by wireless sensor networks, the similarities between WSNs and on-chip networks have been observed in the literature [138].[5] A subsequent C-language implementation, galsC [45], enabled the TinyGALS model to be realised on limited embedded systems. The GALS model is particularly appropriate for embedded distributed systems such as WSNs and MPNoCs as each processing element has a well defined local timing source (such as a clock signal) but processors do not have access to a common clock or 'tick'. *Event-driven* programming is favoured on power and resource constrained systems as it can be implemented very inexpensively. The TinyGALS runtime scheduler was demonstrated to only require 112 bytes of memory [46], and a similar schedular in TinyOS [50] (a minimal operating system for very limited devices such as WSN nodes) only requires 86 bytes [46].

2.3.2.3 HyMR

HyMR [158] demonstrates an effective implementation of the *MapReduce* [52] programming model on non-coherent many-core architectures, such as the SCC. MapReduce is an approach for managing computation on large scale datasets. MapReduce computations happen in two logical operations: the input data set represented as a key-value dictionary is *mapped* to a new list of key-value pairs. Every entry in the input dictionary can be processed in parallel without data dependencies on any other entries, making it particularly amenable to distribution across systems without shared memory or where the entries in the input dictionary are already distributed. The second stage of a MapRe-

[5] Moritz et al. also recognise a similarity to *Service Oriented Device Architectures* (SODA) [51].

duce is, naturally, a *reduce* function. All values with the same key in the output of the map stage are *reduced* into a single value which is the output for that key. Each of these reductions can also happen in parallel without data dependencies. The HyMR implementation is able to effectively exploit non-coherent platforms but is still dependent on the availability of shared memory.

2.3.3 Summary

A wide variety of programming frameworks and models intended to exploit MPNoC and MPSoC architectures can be found in the literature. Message-passing programming models are a natural choice for communication-centric architectures and these are well represented. MPI-like programming interfaces are particularly popular in this domain but issues related to the large size of MPI implementations have been encountered repeatedly. Furthermore, MPI-like interfaces only provide an appropriate abstraction for communication. Other issues, such as the management of tasks and their mapping to available processing resources are not addressed by MPI. Operating systems and low level software to manage MPNoCs [124, 39] and MPSoCs [220] have been considered in the literature, but a platform-independent combination of programming framework and supporting system software has not been demonstrated. The existing approaches suitable for programming more limited architectures are almost exclusively based around the C programming language (or similar variants) and do not provide support for applications with dynamic task structures.

Intel's SCC MPNoC has proven to be an enabling technology that encourages a variety of approaches for appropriate programming models for non-coherent multiprocessors. The SCC *does* still provide a shared-memory architecture and this enables software design patterns that will struggle to scale as the processor count is increased in the future.

2.4 Actor-Oriented Design and Programming

There are a number of computational formalisms that have been used as the theoretical foundations for the programming approaches described previously, and in this context –communication-centric and distributed systems– the most important of these models include:

- *Communicating Sequential Processes* [83] (CSP), which provides the foundation for Occam's model of concurrency, which in turn provides a model for the XC and Handel-C languages.

- *Kahn Process Networks* [107] (KPN), which provides a communications model for the HeMPS programming framework. More generally KPNs are attractive for many systems as they provide complete system-wide determinism.

- *Bulk Synchronous Parallel* [210] (BSP), can be used to model for the 'reduce' phase of a MapReduce computation [154]. BSP has also been used as the foundation for programming extremely large scale hierarchical multiprocessor systems on chip [109].

- The *Actor model* [5, 81, 23], discussed briefly in section 1.4, forms the foundation for a variety of distributed programming frameworks (discussed in section 2.4.1) and has been used to model many different kinds of systems:

  - embedded systems [118]
  - heterogeneous systems [62]
  - real-time systems [233]
  - system-on-chip hardware [92]
  - and networks-on-chip hardware [93]

  However, a recognisable actor-oriented framework for *programming* communication-centric architectures does not exist in the literature.

As discussed previously in section 1.4, the actor model of computation considers a system to be made from isolated *actors* that interact via asynchronous message passing. A subtle but important detail of the actor model is that it does not require messages to be received in the same order in which they were sent. As the model allows for arbitrary reordering of messages, a particular message ($m1$) sent to an actor could arrive only after all other messages that will ever be sent to the actor have been received. Essentially, $m1$ has been lost. In this way the actor model is particularly tolerant of implementation-defined non-determinism and hardware failures.

The intent of the actor model is to provide a theory of computation that is consistent with a 'physical intuition' of the computations that occur in nature [23]. This is the reason for the actor model's asynchronous nature as it is argued that global and consistent

timing does not make sense in a truly distributed system as this concept is fundamentally unrealisable in the physical world [5]. An inherent characteristic of the universe known as *time dilation*, a consequence of special relativity [56, 61], dictates that the rate at which time progresses has no consistent universal definition but is determined by an observer's particular relativistic frame of reference. In the context where an actor is the observer, the passage of time is always constant from its own perspective. However, the apparent passage of time for other actors is dependent on the relative motion of the actor's realisations, and on the proximity of either actor's realisations to a significant gravitational mass. The Global Positioning System (GPS) is an example of a real distributed system that has to account for the effects time dilation [12].

Neither the fundamental asynchrony nor non-determinism of the actor model are *guaranteed* to occur in an implementation but the model allows for this possibility. Agha [5] described basic constructions that can be used to assure message ordering (using sequence numbers) and synchronous behaviour (using a centralised synchroniser actor). However, these constructions do not necessarily improve a system and Agha argues that attempts to synchronise will necessarily create a bottleneck and introduce inefficiencies.

2.4.1 Actor-Oriented Frameworks

A wide variety of actor-oriented programming languages and modelling frameworks exist, with some notable examples including Erlang [104], CAL [222], Scala [77], Akka [91], Kilim [193] and Ptolemy II [3].

2.4.1.1 Ptolemy II

Unlike the others which are programming languages or frameworks, Ptolemy II [3] is a system-level modelling framework that enables systems to be represented as a graph of connected 'actors'. A distinguishing feature of the Ptolemy approach is that the execution semantics of a system are not a fundamental characteristic of the Ptolemy tool, but instead the execution semantics are selected by a *director* that is embedded in to the design. This enables Ptolemy to effectively model many different *Models of Computation* (MoCs) including continuous time (CT), discrete event (DE), process network (PN), dataflow (SDF) and a CSP-like rendezvous model. While Ptolemy II provides a powerful modelling environment for static arrangements of actors, it is not well suited to mod-

elling dynamic architectures and is not intended to be an actor-oriented programming framework: there is no reliable path from a Ptolemy II model to standalone executable code, although there have been several approaches presented for limited code generation [208, 189, 119, 212].

2.4.1.2 Programming Languages

The perceived scalability and fault tolerance offered by the actor model are the primary motivation for the construction of actor-oriented programming frameworks. In practical implementations the programming model is not *purely* actor modelled but a hybrid of the actor-model for expressing the principles of concurrency and some other paradigm is used to express sequential computation and state within each actor. An important characteristic for an actor-oriented programming model is therefore the conceptual division between the two (or more) different paradigms that it draws upon. In the case of Erlang [104] this division is between a *functional* programming paradigm for sequential code and recognisably actor-modelled concurrency:

- Erlang's units of concurrency (called *processes*, but not OS processes) cannot share any data, a so-called *shared-nothing* (SN) architecture.
- Erlang processes can *only* [104] communicate via message passing.
- Each process contains a single *mailbox* which receives messages sent to it by other processes.
- Sending messages uses a Occam style '`!`' operator: '`Pid ! Message`' sends the `Message` to the process identified by the value of the `Pid` expression.
- Unlike Occam channel sends, Erlang's send operation never waits; messaging is entirely asynchronous.
- Processes use a `receive` primitive to express how messages will be handled. Pattern matching is used predicate behaviour based on the type of message received.
- Messages are stored in a process's mailbox in the order that they arrive, but the pattern matching mechanism allows a process to specify the order in which messages will be retrieved from its mailbox.

- Process identifiers are provided when a new process is created with the `spawn` built-in function (BIF) and can be included in messages just as any other value can.
- Process identifiers can be bound (`register`) to system-wide names enabling them to be looked up (`whereis`) by other processes. A process can also enumerate all registered names by retrieving a list from the `registered` function.
- Process failures are isolated and have no effect on unrelated processes. However, processes can be *linked*: When a process fails an `EXIT` signal is sent to all of its linked processes. Process links are bidirectional and the default behaviour provided by the language is for a process to terminate if it receives an `EXIT` signal.

Internally, Erlang processes are lightweight threads and do not require their own OS threads. This is also the approach taken in Akka [91] and Kilim [193] as it vastly reduces the overheads needed for each actor. Akka retains the model of a single mailbox per actor but an actor implementation can chose between different mailbox semantics by implementing a particular **`interface`** (in the Java version). Akka mailboxes can be bounded, and message arrival order (message priorities) can be specified. Akka allows *typed* actors to be defined that can only receive messages of a particular type, but the consistency of actor typing cannot be verified statically as actors can be located at runtime via paths represented as strings. Just as Erlang provides a process directory, Akka also enables actors to be dynamically enumerated regardless of their implementation or location in a distributed system.

Akka actors are somewhat less isolated than in Erlang or Kilim. Kilim actors have statically enforced isolation that ensures all actors have disjoint heaps. Kilim's use of *isolation-types* [194] ensures exchanged messages are only ever accessible (aliased) by a single actor. This formulation enables mutable data to be sent via mailboxes. In contrast Akka does not enforce strong isolation of its actors: immutability or limited aliasing is not enforced for messages and code can be constructed to modify the internal state of other actors. Akka actors have to carefully guard references to their object (the **`this`** reference in Java) to prevent modification by other actors executing in the same virtual machine. Correct Akka applications are required to use special *proxy* references to pass their reference another actor. In addition, Akka actors are implicitly linked to the actor that created them, with the parent actor being known as the *supervisor* actor. In contrast

to Erlang, Akka's supervision (linking) structure is only allowed to be tree-shaped.

Kilim differentiates itself from Akka and Erlang by decoupling mailboxes from actors: mailboxes are treated as any other Java object that can be exchanged between the actors, and an actor can have access to any number of mailboxes. Similarly the *capsule-oriented* programming approach [178] effectively allows multiple mailboxes per 'capsule' but these are represented as method invocations rather than explicit mailboxes; distribution of capsules is hidden from the application entirely.

2.5 Java

Java [74] is a popular general-purpose, object-oriented programming language. It is differentiated from other modern general-purpose languages by the combination of a high degree of portability between architectures and operating systems and its strong static typing system. In addition to these qualities most implementations of Java also benefit from entirely automatic memory management, but this is a common feature in modern programming languages [105, §1.2]. Java is most often compiled to bytecode that executes [74] on its own *Java Virtual Machine* (JVM). Java itself is not directly related to the problem domain, but forms a sufficiently important component of the work presented in Chapters 4 and 5 that some background is warranted.

Standard Java is not designed for programming resource constrained distributed systems but it has several characteristics that make it more suited to this environment than C, including:

- Java makes no explicit von Neumann [213] architecture assumptions: Java code (encapsulated in *classes*) does not necessarily have to reside in the same storage as data memory.
- Java does not make any assumptions about how memory is organised in the host architecture, for example there is no requirement that memory is integrally addressed. All data is stored in *objects* which may only access one another by opaque references. These references only have meaning to the virtual machine executing the Java bytecode and may not be manipulated as in C (by using pointer arithmetic, for example).
- Java has concurrency support as part of the language. A `Thread` [74] class is used to create new concurrently executing threads of control and all objects in Java

contain a *monitor* [82] to allow synchronisation and mutual exclusion directly at the language level.

- Java is designed to manage the complexity of large systems by providing concepts such as packages and access control for data.

These points do not necessarily make Java suitable for resource constrained implementation but *less unsuitable* than C. Perhaps because Java is not profoundly expensive to implement and because of its popularity, a large array of embedded and reduced Java implementations have been presented. A limited selection of these approaches are presented below.

2.5.1 Java in Resource Constrained Contexts

Varma and Bhattacharyya [212] detail a Java-to-C converter to enable the an application written in Java to be compiled 'though' a C compiler to native code, sparing the overheads of a full Java runtime environment. The use of C [110, 97] as an intermediate language is a natural choice; practically every processor, microcontroller and DSP will have a functional C compiler. In addition to being widely supported, C compilers such as the popular GCC [67] have sophisticated optimisations built-in to minimise execution time and binary sizes without the need to build these optimisations into the Java translator.

Java has been applied successfully to environments with as little as 512 bytes of RAM and 4KiB of program code through the use of the Java Card [196] specification, or the JEPES [190] platform. However, both the Java Card specification and the JEPES platform make significant changes to the Java environment in to enable execution on such limited hardware. In particular both of these approaches change the available standard libraries and reduce the set of available primitive types available to the programmer: Java Card simply forbids several primitives including **`char`**, **`float`** and **`double`**. JEPES makes the existence of floating point types dependent on the target platform and reduces the widths of other primitive types too.

In contrast to the use of a specifically cut-down Java specification, Varma and Bhattacharyya [212] enable the use of the full Java libraries by extensively pruning unused code from application and libraries under translation such that only used code is present in the resulting C code. Toba [172] and GCJ [68] which are not intended for use in

embedded systems do not perform such code pruning. Toba, which is no longer maintained, used a strategy of elaborating Java's full class libraries into C. Translating a whole Java library greatly simplifies compilation and saves time as this only need be done each time the Java libraries are changed. The compiled Java library can then be linked against the user's own code without any special consideration. This is also the approach used by GCJ which requires that the user's code be linked against a large 'libgcj.so'. However, as of GCJ 4.4[6] this shared object is 34MiB putting it well out of reach of the capabilities of small embedded systems.

In spite of the aggressive code pruning employed by Varma and Bhattacharyya [212], the costs associated with the dispatch of virtual methods are not addressed and the runtime in-memory class descriptor structures are especially expensive (including class names, references to superclass structures, the size of instances, and if the class is an array or not) considering that reflection is not supported by their compilation process.

In another similar work, Nilsson [142] presents a Java to C compiler with an emphasis on support for hard real-time applications. Importantly, their work incorporates the implementation and evaluation of real-time garbage collectors suitable for including on a sophisticated (at the time of publication) microcontrollers such as the Atmel ATmega128 [15]. However, Nilsson reports disappointing performance while the garbage collector is enabled. Nilsson also confirms the observation that the virtual dispatch of Java methods can have a significant impact in runtime performance (they observe a 43% increase in execution time in one experiment when dynamically dispatched methods are used compared the same experiment with statically dispatched methods). While Schultz et al. [190] in the JEPES work also recognise that dynamically dispatched methods come at a significant runtime price, this is accepted to be an unavoidable cost.

2.5.2 Java in Hardware

Another technique for enabling Java in embedded systems is to create a processor to natively execute Java bytecodes. JOP [187] is an implementation of a Java virtual machine that can be synthesised for use on an FPGA to provide time-predictable Java bytecode execution. As Jop provides a processor that is itself capable of running Java bytecodes

[6]While not officially discontinued, GCJ is effectively out of development. The most recent update was in March 2012 with the previous update in 2009. OpenJDK [151] has largely rendered the GCJ and GNU Classpath redundant.

without a software implemented virtual machine to interpret the Java instruction set, it can be correctly called a 'Java machine'. Unlike Sun Microsystem's picoJava [146] hardware Java machine, Jop only implements a real-time subset of the *Java 2 Micro Edition* [134] (J2ME) Java specification. Java virtual machines have been implemented in hardware many times with different design goals [174, 78, 101] indicating the plausibility and draw of this approach. Where an embedded system will be implemented on an FPGA the use of a true Java machine is attractive as it simplifies the tool-flow from Java code to execution in the target system. However, the use of a Java softcore does not necessarily help with the problem of unsuitably large Java libraries.

Where there is the ability and appetite to fully redefine the implementation platform, the direct synthesis of Java to digital logic for FPGAs has also been demonstrated several times: Sea Cucumber [207] and Galadriel [41] are two approaches that have the advantage that application-specific hardware can be described in Java itself, but the supported subset for synthesis is very limited. Neither approach supports the manipulation of objects from synthesised Java methods, exceptions are not supported, recursion is forbidden, and a Java method synthesised into a hardware accelerator is not allowed to call methods in software.

2.6 Summary

Elements of the actor-oriented approaches discussed in section 2.4.1 are attractive for programming communication-centric architectures:

- Strong isolation of actors in time (asynchronous execution) and space (data confinement) avoids a dependence on shared memory and common timing sources.
- Type-safety of actor communications helps to manage the complexity of applications by detecting inconsistencies.
- Multiple mailbox semantics, including the ability to bound a message queue's length, is desirable when memory is extremely limited.
- Event-driven applications are amenable to lightweight implementations. OS-level processes and threads are not required.
- Dynamic application structures that can respond to hardware and software failures, as well as changes in workload, can be expressed.

However, the existing actor-oriented frameworks also present a number of challenges:

- Erlang and Akka's ability to enumerate actors introduces centralisation.
- Akka allows actors to be addressed by their physical location, creating platform dependence in an application.
- Kilim's decoupled mailboxes create a confused model: actors can essentially swap addresses by exchanging mailbox objects, and mailboxes can have lifetimes independent of actors. This model is an interesting middle ground between rigid models where actors are their own addresses and models where mailboxes are first class objects that can be freely shared between actors.
- The sophistication and complexity of these frameworks indicates a significant challenge for the exploitation of resource constrained architectures.

These observations suggest the opportunity to define and explore a more appropriate programming model for resource constrained MPNoCs that can build upon the strengths of the previous actor-oriented approaches.

Chapter 3

Machine Abstract Architecture

Contents

2.1 Multiprocessor Networks on Chip . . . 33
2.2 Embedded System Design . . . 35
2.3 Programming MPNoCs . . . 38
2.3.1 Hardware-Specific Approaches . . . 40
2.3.2 General Purpose Communication-Centric Programming . . . 44
2.3.3 Summary . . . 46
2.4 Actor-Oriented Design and Programming . . . 46
2.4.1 Actor-Oriented Frameworks . . . 48
2.5 Java . . . 51
2.5.1 Java in Resource Constrained Contexts . . . 52
2.5.2 Java in Hardware . . . 53
2.6 Summary . . . 54

In this chapter an actor-oriented programming model is explored to enable the exploitation of resource constrained MPNoC architectures. The description of another programming model is motivated by the difficulties of programming such architectures using conventional thread-based concurrency and the limitations of the models discussed in chapter 2. The exploration of alternative programming models and existing approaches for NoC programming revealed a variety of further areas for potential improvement:

- Static application structures inhibit scalability and portability. Without the ability to dynamically construct actors a programming framework cannot respond to changes in application or environmental requirements. Fault tolerance is also substantially impaired as failed actors cannot be instantiated.
- Strong communications synchronisation is unnecessary and expensive.
- Inflexible communications models do not allow applications to participate in the reliability-performance tradeoff.
- Implicit supervisors, persistent actor addresses and actor registration can introduce centralisation.

The trends in silicon technology identified in chapter 1 and the qualities of existing MPNoC architectures discussed in chapter 2 lead to the following additional observations:

- There is an ever increasing abundance of parallel processing resources but with fewer guarantees of inter-processor synchronisation or the availability of cache-coherent shared memory.

  This suggests that a successful application model should gracefully accommodate an increase in concurrent processing resources, and should avoid a requirement on memory sharing at all costs.
- There are compelling justifications for the existence of resource constrained processors, including cost reduction and further increasing parallelism within a fixed area of silicon.

  This suggests an application model should prefer simplicity over sophistication. An application can implement additional sophistication if required, but an expensive model will invalidate its use on whole classes of architectures.

- The overall complexity of software applications will continue to increase, motivated by the increases in hardware capacity to support additional complexity.

  This suggests a successful application model will facilitate automated tooling and software engineering techniques. There is tension between the static definition of an application's structure, which is helpful for analysis, and its ability to scale indefinitely at runtime. From the first observation, the application model should favour indefinite scalability over fixed application structure definition.

Altogether this allows a list of desirable characteristics for a programming model in the context of resource constrained MPNoCs:

portability describes the cost of re-targeting the application to operate on hardware other than it was originally designed for. The ideal porting cost for an application is zero, and while this is often unlikely to be achievable, minimising application porting costs is critical. It is realistic to expect that there will eventually be a bewildering variety of different highly multiprocessor communication-centric architectures. The success of a programming model will depend on its ability to insulate programmers and engineers from the architectural differences while enabling efficient execution.

scalability describes the cost of growing a system to accommodate greater demands on its functionality. While the ideal cost of scalability is zero (i.e. indefinitely increasing the capability of a system without any additional cost), this is not possible. Therefore the goal is that the costs for scaling (porting) an application from a small platform to a large one is very low or free, and that the inherent overheads of implementing the programming model grow no faster than linearly with respect to the 'magnitude' of the platform. In this context the magnitude of a platform would be some function of the processor count and their capabilities, and the characteristics of the interconnection fabric.

practicality A successful programming model must facilitate rather than hinder application development. More specifically this means that it must work, and all additional requirements imposed on application designers must be justified. A substantial fraction of this requirement is implementation dependent.

efficiency is almost entirely implementation dependent, but the model must also avoid creating requirements that are expensive to implement. This is the least important

of the desirable characteristics as if the first three can be substantially satisfied then future effort to improve efficiency will be well justified.

3.1 Machine Abstract Architecture

The *Machine Abstract Architecture* (MAA) is a template for computer systems including a model of the software, the hardware and the relationships between of the two. The application model, which is the primary focus of this chapter, aims to address the four desirable model characteristics (portability, scalability, practicality and efficiency) such that they are optimised for MPNoC platforms.

In this chapter the discussion of concrete architectural elements, such as specific processor architectures or programming languages, is avoided as far as possible. The purpose of the Machine Abstract Architecture is to help communicate clearly the relationship between the various abstract system concepts, and to provide a framework in which to discuss the subsequent topics.

3.1.1 Systems

Systems are instances of the Machine Abstract Architecture and are composed of an *Application* (discussed in section 3.2) to represent the structure and behaviour of the software, and a *Platform* (discussed in section 3.3) to represent the structure and characteristics of the hardware resources. The MAA also specifies a 'framework' element which defines the limited coupling of the application and platform elements of a system discussed in section 3.4.

This high-level definition is illustrated in figure 3.1. An acceptable alternative interpretation is that the application layer reflects the portion of the system that is currently being defined, while the platform section of the system has already been fixed. This interpretation allows the possibility of reasoning about custom hardware accelerators and hardware-software co-design via a machine abstraction, but the primary focus of this thesis is on *programmable* multiprocessor platforms and as such applications will usually considered to be software.

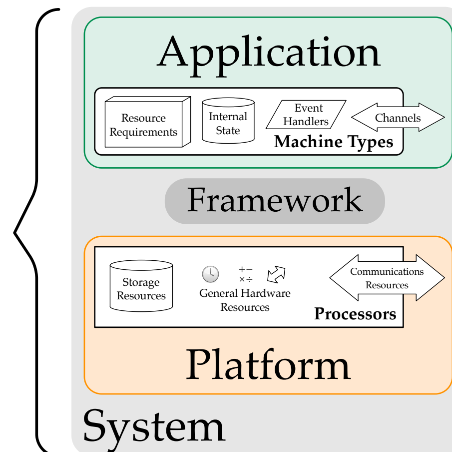


Figure 3.1: *The definition of the Machine Abstract Architecture at the highest level: A system is an application and a platform. The 'framework' element describes the coupling between the two.*

3.2 Application Model

Within the machine abstract architecture, *machine oriented* programs are structured according to the *application* model. The application model is an extension of the actor-oriented model of computation: All behaviour in a machine oriented program results from the execution of many independent *machines* which encapsulate both the data and computation aspects of software. Every machine executes concurrently without any common notion of time shared with other machines. A machine only has access to its own memory and resources. Interaction between machines is limited to message passing via statically defined channels.

In this section the application model of the machine abstract architecture is discussed, including:

- how an application is defined (below)
- the properties of a machine (section 3.2.1)
- how machines interact (section 3.2.5)

Machine oriented applications are defined by a collection of machine specifications (*machine types*) where one particular machine specification is denoted as the *start machine*. Instances of these machine specifications are the actors of the system at runtime, but only the start machine type is *defined* to have an instance at runtime. When a system begins to execute a machine of the start type is created on the *first* processor (see section 3.3) of the platform. The static definition of the application does not include which instances of

other machine types will eventually exist. At runtime a machine can cause any quantity of any type of machine to be created. Once created machines can react to the receipt of messages, the passage of local time, and to their own creation. On each of these events the machine can perform arbitrary computation, possibly involving the transmission of messages to other machines, updating its own internal state or creating new machines.

An example of an application structured in a machine oriented style is provided in figure 3.2. In this figure each machine *type* is one of the white rectangles. The channels (think 'message boxes') defined by the machine type are the large arrows escaping the machines to the right. The direction of *data* flow is indicated by the channel arrow's direction, except in the case of the 'Stimulus' channel shown in the **Emergency Valve** machine which neither transmits nor receives any data. It can be seen in the example that the **Flow Controller** type is the defined 'start' machine type and so will be the first machine to exist at runtime. Further facts about the runtime structure of the application can only be obtained through analysis of the event handlers and internal state (if defined) for each machine type:

- The application model does not define the structure of a machine type's internal state but this detail has been provided in figure 3.2 for clarity. Implementations of the machine model expressed in strongly typed or object oriented programming languages will naturally express more information about the structure of a machine's internal state. For example, the internal state of the **Level Sensor** machine type in figure 3.2 contains machine references to **Flow Controller**, **Unreliable Sensor**, and **Emergency Valve** types. This allows the inference that the **Level Sensor** has runtime dependencies on these types: It might create, communicate with, or receive references to machines of these types.

  However, as the application model only specifies that internal state exists and not the specification of its structure, application analysis via state structure cannot be guaranteed across implementations.

- As with the internal state of a machine, the exact behaviours and syntax of event handlers are not defined by the application model; they are implementation defined and within the confines of a single machine the behaviour of an event handler is immaterial to the model. In figure 3.2 the behaviour of the event handlers is described informally in English.

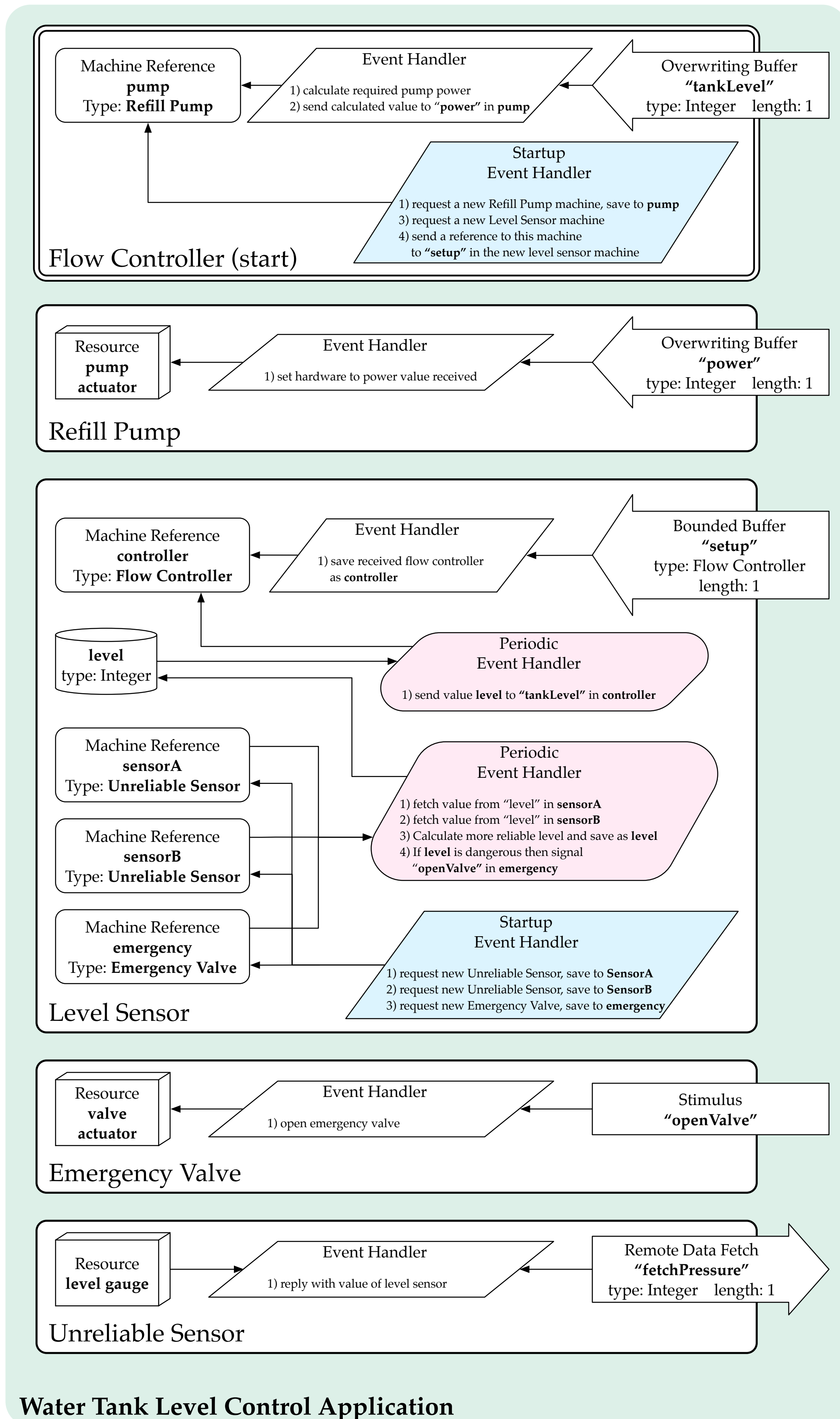


Figure 3.2: *The definition of a machine oriented water tank level control application.*

However, the application model does impose certain constraints (discussed in section 3.2.6) which enable some structural details to be extracted from event handlers. In particular it can always be determined which types of machines a machine *may* interact with, and which channels of other machine types the event handler *may* use. The 'may' caveat alludes to the consequences of the halting problem [128, 209], and more specifically Rice's theorem [182], which states that there is no general and effective method for determining non-trivial properties of algorithms. In the context of event handlers this means that it cannot *in general* be determined what the outcome of the execution will be. The presence of a named channel within an event handler is an indication that the code *may* use that channel but it cannot be guaranteed *for all possible event handlers* if the channel would actually be used, or how many times if it is used. However, it is certain that a machine *cannot* use a channel if it is not referenced by an event handler.

Although it is impossible to guarantee to determine exact properties of *all possible* event handlers, real program code is likely to be amenable to forms of static analysis enabling more detailed program structure to be extracted from an application definition.

Figure 3.3 highlights the properties that the model guarantees will be extractable from an application. The extractable information is only sufficient to generate graphs of dependencies between the machine *types*. A summary of the static machine dependency graph for the water tank example application is provided in figure 3.4. It can be seen that this information is not enough to determine statically that there will be two instances of the **Unreliable Sensor** machine at runtime. The **Unreliable Sensor** machine type requires a **level gauge** resource so it is possible to infer that there cannot ever be more **Unreliable Sensor** machines than the platform has **level gauge** resources. The topics of resource usage and provision are discussed further in sections 3.2.7 and 3.3.2, respectively.

An entity-relationship diagram for the application model is provided in figure 3.5. The semantics of a machine type are defined by its event sources and handlers. *Event sources* can be considered to be the 'interface' of a machine as this defines which channels a machine type exposes to other machines, likewise *event handlers* are the 'code' of a machine. All computation and behaviour of a machine is defined within the machine type's event handlers. Each instance of a machine type has distinct state and operates

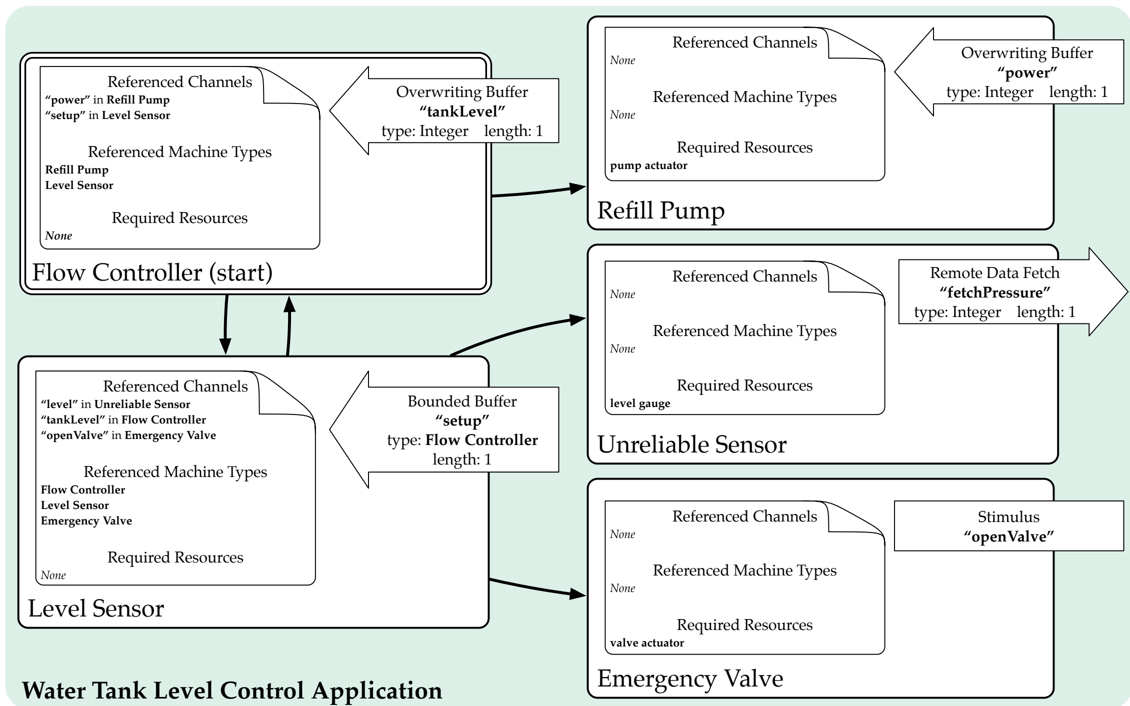


Figure 3.3: *The properties that are guaranteed by the application model to be extractable from the example application. The black arrows indicate where a machine type has a direct 'communicates with an instance of' relationship to another type.*

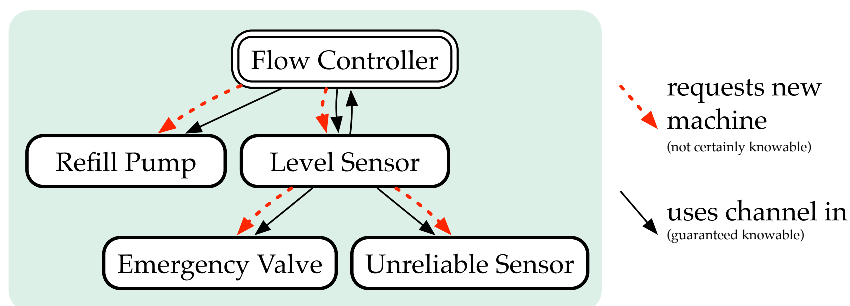


Figure 3.4: *The machine dependency graph obtained from the example application.*

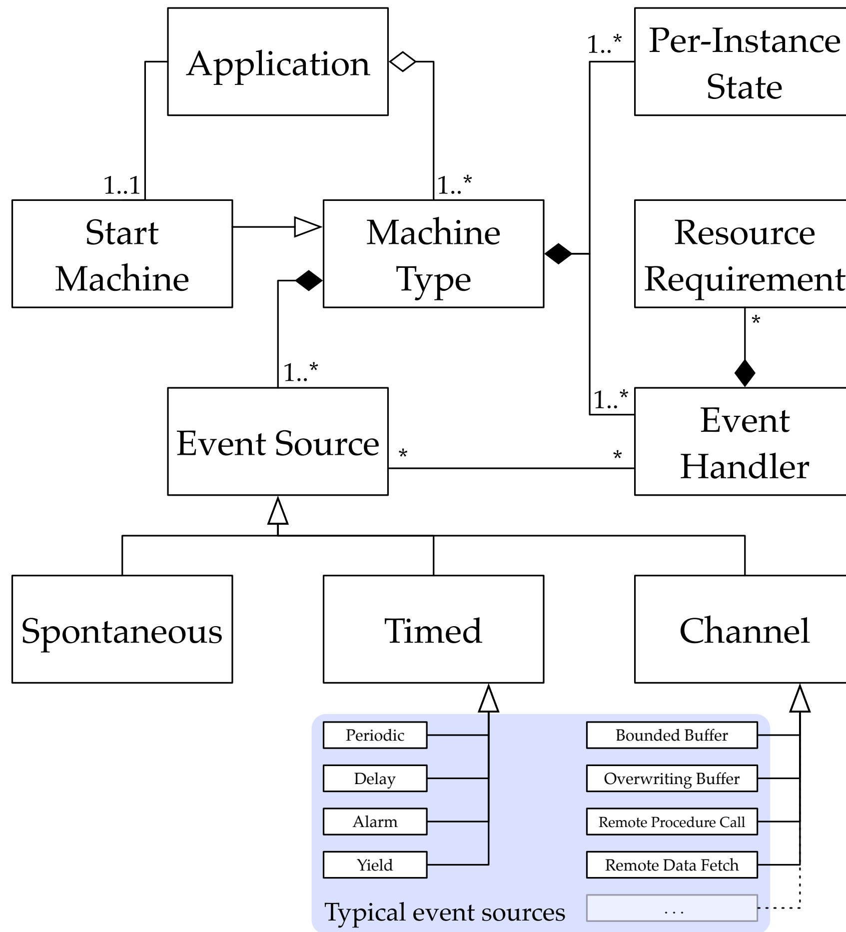


Figure 3.5: *An entity-relationship diagram for the application model.*

independently to all other machines in the application.

The definition of an application is primarily intended to describe the behaviour of a system but structured in a way that is amenable to analysis and implementation on diverse platforms. As such, all implementation strategies are equally valid if they can produce the runtime behaviour that matches the definition of the application. This means that elements of the application definition do not have to exist at runtime if they are not needed for correct runtime semantics. For example, if a tool is able to map certain machine types onto hardware function accelerators when their behaviour matches, then there may be no need to generate any code or metadata for those machines. Even when machines are executing on general purpose processors it may be possible to generate entirely static code that has no dependencies on the defining structure of the application.

3.2.1 Machines

Machines are the central concept in the machine abstract architecture: Each machine is the combination of an execution context and a container for state. Runtime execution of code only happens in the context of a machine and all machines in an application are potentially executing concurrently. There are no other 'threads of control' in an application. In a similar fashion all application state is contained within exactly one machine; there is not any state shared between machines nor any state which is owned by no machine. The isolation of machine state is much stronger than the typical access modifiers (**`public`**, **`private`**, **`protected`**, etc.) provided in some programming languages. Machines *never* have access to each other's state, even if they are instances of the same machine type.

Every machine at runtime is an instance of a defined *machine type*, and these machine types are the primary constituents of the application definition (see figure 3.5). There can be any number of machines of each type at runtime, in practice the number is only limited by resources available in the system's platform. An application where the machines have no specific resource requirements (i.e. where only processing, memory and inter-machine interactions are required) would have no defined limits to its magnitude at runtime on a hypothetical unbounded platform.

A well designed application would be able to become arbitrarily large at runtime as machines have no spatial or temporal dependencies on one another, and do not even have any knowledge of other machine's existence in the application. Machines are *local*

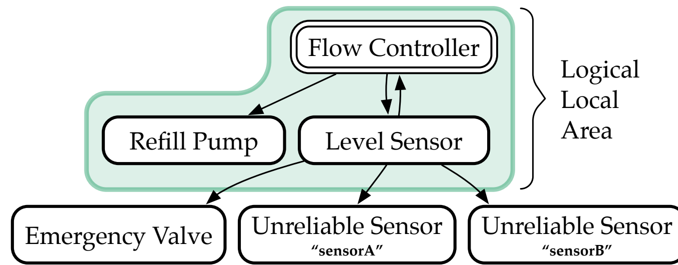


Figure 3.6: *The runtime machine instances of the water tank example application. The logical local area of the* ***Flow Controller*** *machine is represented by the green shaded region.*

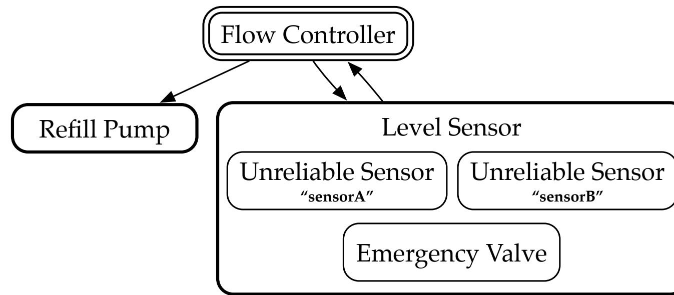


Figure 3.7: *The* ***Level Controller*** *machine essentially encapsulates the* ***Emergency Valve*** *and* ***Unreliable Sensor*** *machines; it* hides *this complexity from the* ***Flow Controller*** *machine*

entities: All interaction between machines, including synchronisation, only happens through the application's defined channels, and these can only be used by a machine if it owns a valid *reference* to the other machine. A machine reference is an opaque identification of a particular instance of a machine type at runtime and all references to the same machine instance are identical. As an application definition does not specify any machine instances, it therefore also does not contain any machine references. A machine has permanent references to itself and to a platform-defined set of remote *ProcessorManager* machines (see 3.4).

References to any other machines can only be obtained by:

creation of a new machine. The machine which requested a new machine will have a reference to the newly constructed machine. See section 3.2.4.

communication of the reference from one machine to another. Machine references are valid data types for intermachine communication. See section 3.2.5.

The critical detail is that machines are not able to *synthesise* references to other machines and there is no built-in mechanism to index or enumerate other machines in an applications. A machine can only interact with the *logical local area* defined by the union of the set of references currently part of its state, and the permanent references made available

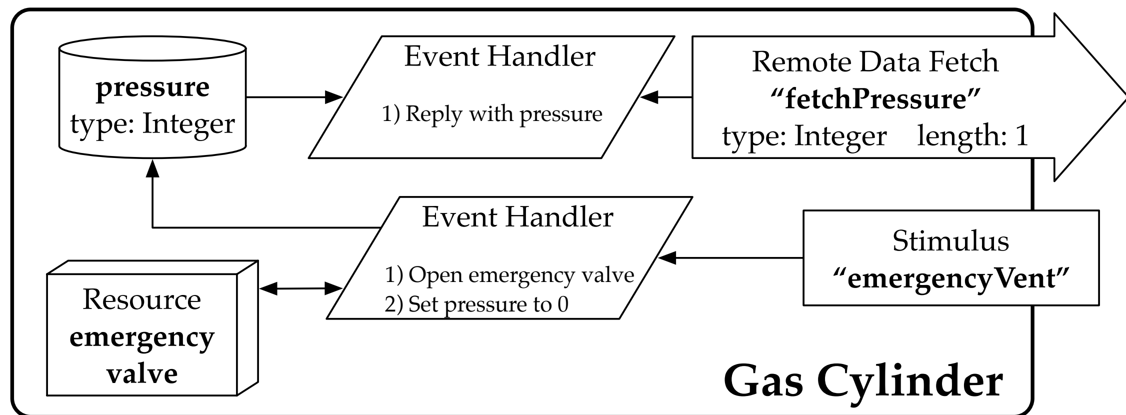


Figure 3.8: *A simplified gas cylinder defined as a machine. This type of machine definition expresses the connection between a specific gas cylinder's level and emergency vent capability.*

to it by the runtime framework. In the previous water tank example application (see figure 3.2) the **Refill Pump**, **Emergency Valve**, and **Unreliable Sensor** machine types contain no machine references as part of their local state and so have the smallest possible logical local area, potentially only composed of their own machine. The internal state of the **Flow Controller** machine has the logical local area illustrated in figure 3.6. It can be seen in figure 3.6 that the **Level Sensor** effectively *hides* its internal implementation from the **Flow Controller** machine, which has no awareness of the existence of the **Unreliable Sensor** machines which are used to perform the level measurements. As only the **Level Sensor** machine ever has references to the **Emergency Valve** and **Unreliable Sensor** machines they can be considered to be *encapsulated* by the **Level Sensor** machine type. This structural encapsulation is shown in figure 3.7. Although **Emergency Valve** and **Unreliable Sensor** may be considered structurally encapsulated by the **Level Sensor** machine, they have the same timing and spatial independence from **Level Sensor** as every other machine does.

In contrast to other actor models, machines types can define any number of communications channels including zero. The *aggregation* of multiple channels by a machine type provides an unambiguous indication that the channels have a relationship to each other. This relationship is not expressible in computational models where actors have at most one inbound channel or where channels can exist as standalone entities. The machine type shown in figure 3.8 expresses the important relationship between its two channels. An application containing instances of the **Gas Cylinder Machine** could always be certain that if a pressure level was fetched from a "fetchPressure" channel, then the coupled "emergencyVent" channel would correspond to the same cylinder in-

stance. Without channel aggregation an application could never be certain that multiple channels intended to have a shared 'context' actually refer to the same context as one another; channel aggregation prevents compatible but unrelated channels from becoming confused. Channel aggregation therefore exposes much more application structure and can prevent a class of programmer errors.

3.2.2 Timing, Scheduling and Synchronisation

The isolated nature of machines means that the *inter*machine synchronisation considerations are somewhat different to the *intra*machine timing and scheduling considerations. Machine oriented applications can be viewed as an instance of the of the *globally-asynchronous locally-synchronous* (GALS) [43, 139] architectural pattern as each machine is internally synchronous, but there is not a synchronous relationship between machines.

3.2.2.1 Intermachine Synchronisation

At runtime all machine instances operate concurrently and without any *defined* timing relationship to other machine instances. This allows total freedom for implementations to allocate multiple machines to a processor, dedicate processing resources to a single machine, or even implement machines as dedicated special purpose functional units. Implementations are also not constrained by the application model to provide a global timing source. Therefore application code cannot make any assumptions about the execution speed of code in any other machine, even if the other machine is the same type. Undefined and hidden timing relationships between machines are almost certain to exist when an application is implemented on any present day computing architecture from embedded MPSoCs to warehouse scale HPC, but these relationships cannot be relied upon by a portable application.

Some types of communications channels, such as a *rendezvous* (see section 3.3.3), can provide inter-machine temporal relationships but such relationships are fundamentally short-lived. Inter-machine synchronisation only lasts for the duration of the transaction that established the relationship and once concluded the machines are once again decoupled in time. It is not only possible but inevitable that machines will lose synchronisation when the implementation platform does not have a global timing source, as all physical timing sources will have imperfections that guarantee they will diverge over time.

As machines are mutually asynchronous, both continuous synchronisation of machines' behaviours and determining the order of events across the system are challenging. *Synchronizer* [19] algorithms can be used to simulate a synchronous system on an asynchronous substrate but these introduce a substantial communications overhead. The issue can be somewhat mitigated by avoiding a requirement on machine synchronisation and instead using the concept of *logical time* to determine the order of events within an application and to make distributed decisions. Any logical timing scheme such as Lamport timestamps [114] or vector clocks [130] can be implemented within the framework of the application model.

3.2.3 Intramachine Timing and Scheduling

At runtime all code in a machine is contained within event handlers which are executed in response to events triggered by the machine's event sources. The application model considers event handlers to be black boxes and so does not define the execution paradigm or timing aspects of event handler code; the semantics of a particular event handler's code is the concern of the implementation language.

The most important aspect of a machine's internal execution is that it is *sequentially consistent*. Sequential consistency is the requirement that the execution of code happens with the same consequences as if the code had executed on a single sequential processor [115]. This means that only one event handler may ever be 'live' at once, and therefore also means that machines must execute event handlers *non-preemptively*. If two or more event handlers are live at once, or if an event handler could be preempted by another then this could result in non-sequentially consistent behaviour being observed if the second event handler modified the data structures that were in use by the first.

Together with the guarantee of machine spatial isolation, the non-preemptive and sequentially consistent execution properties mean that application code can be written without any consideration for issues raised by concurrent data access. Concurrent access to data is impossible so there is simply no need for implementations to provide conventional synchronisation primitives such as locks, semaphores and monitors.

Implementations are free to use any form of concurrency or parallelism within the bounds of a single event handler, but no other event handler can begin execution until the first event handler has completed. This means that implementations cannot start 'background' threads that would continue to execute with a lifetime beyond the event

handler that started it.

Machines can contain non-event handler code for implementation specific initialisation purposes if the code meets two conditions:

1. it does not require any input data
2. it has no consequences for anything but the local machine.

This type of program code is unlikely to have any utility other than the algorithmic initialisation of the machine's internal data structures. The result of the execution of this code will be identical every time it is run as the input will always the same.

3.2.3.1 Event Sequencing

Immediately after a machine has been initialised a *startup* event is triggered and the startup event handler is executed if defined by the machine type. This event handler is guaranteed to be the first handler executed in a machine and unlike initialisation code it has no restrictions on its behaviour; it is free to request new machines and perform arbitrary computation. The machine that requested the new machine to exist may already have been provided with a reference to the new machine before the startup event has been handled. This does not present any problem as any event triggered in the new machine will not be handled until after the setup event has completed.

Each machine has its own internal priority queue for events that have been triggered by event sources but have not yet been handled by their event handler. An event source is only permitted to have at most one event in the queue at any instant. This implies the event source must wait until the first event has been handled to enqueue a subsequent event. This restriction bounds the maximum possible length of a machine's event queue to the number of event sources in the machine type. This is highly desirable as it ensures event sources contain their own buffers if needed rather than using the event queue as an implicit buffer. This is better as it allows event sources to define their own buffering semantics including ordering, length and response to overrun conditions.

When a machine is idle it will chose the highest priority outstanding event to handle. The priority of an event is determined first by the priority of the event source that it originated from, and secondly by arrival order. Therefore, if all event sources in a machine have equal priority the machine will handle events in FIFO order. An illustration of this scheduling policy for a machine with four event sources is provided in figure

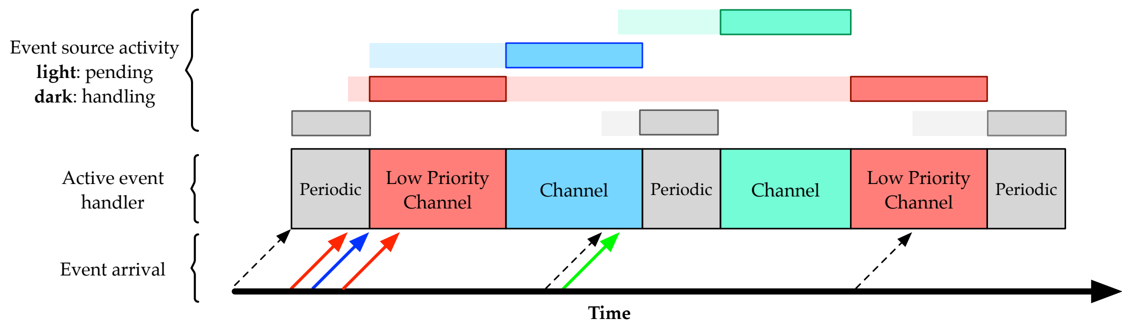


Figure 3.9: *An example of event handler scheduling in a machine with a periodic event source and three channels.*

3.9, where it can be seen that the machine is severely overloaded with almost all events handled late. The application defines the priority of each event source on a per-instance basis. The machine's event queue itself is not visible to application code and may not be required to actually exist at runtime. For example, if a machine only contains a single event source an implementation would only need a flag indicating that an event is pending.

If a machine finishes the execution of its current event handler and the event queue is empty it will remain idle until a new event is triggered. Machines do not have an idle task concept so an idle machine has no behaviour. Idle behaviour can be emulated using a low priority **Yield** event source discussed in the next section.

3.2.3.2 Spontaneous Event Sources

Spontaneous event sources allow a behaviour in a machine to be triggered without an external cause. As previously discussed, intermachine timing dependencies are problematic and cannot be supported, but within the context of a single machine time *is* a well defined concept. It makes sense to define some timing event sources in the application model as they are straightforward to abstract and they enable patterns of spontaneous behaviour to be exposed by the programmer that would otherwise only have been discoverable with sophisticated program analysis. In the language of real-time systems [37, §1], these are *time-triggered* event sources.

There are four primitive spontaneous event sources that can be abstracted easily and are compatible with the machine model:

Alarms trigger an event at a single absolute time in the future. **Alarms** cannot trigger again unless a new absolute time is set. **Alarm** event sources enable *time-aware* [37]

systems to be described.

Delays trigger an event after a fixed interval since starting the **delay**. A triggered **delay** can be started again at any point during or after their event has been handled. **Delays** ensure that the time between starting and triggering is at least the interval specified. **Delays** allow *time-triggered* but *aperiodic* activity to be described.

Periodics trigger an event after a fixed interval since the **periodic** event source last *triggered*. Periodic event sources are restarted automatically after the event handler is completed. In contrast to **delays**, **periodics** aim to keep the interval between event triggers fixed regardless of the execution time of the event handler, or the delay between event triggering and event handling.

Yields trigger an event on request. The **Yield** event source enables intramachine co-operative multitasking. As event handlers are executed non-preemptively, a long-running event handler will prevent other event sources from having their events handled. An event handler that needs execute for a long time can allow other event handlers a chance to execute by finishing early after using a **yield** to add a new event to the machine's queue immediately. As long as the **yield** event source has a priority that is not greater than any other event source this will give all other pending events a chance to execute. **Yield** event sources have the interesting quality that they do not fit into the existing taxonomy of real-time systems behaviour as they are neither *time-triggered* nor *event-triggered*; future **yields** are triggered by the in-progress execution of some code, but the **yield**'s handler execution is only loosely coupled in time to triggering of the event source.

If an implementation provides application code with a 'timestamp' API to access the current time relative to some fixed point, then both **delays** and **periodics** can be emulated with the correct use of an **alarm**. Even if it is possible in a given implementation to emulate the behaviour of **delay** and **periodic** event sources there is utility in making these usage patterns explicit. The representations used for the specification of absolute time and intervals are implementation defined.

An important detail of the timed event sources is that they can only guarantee *minimum* intervals. Long running or non-terminating event handlers can entirely prevent a timed event source from meeting its specifications, meaning that timed event sources can only operate on a "best effort" basis. This problem is not as severe as first ap-

pears though as event handler interference is contained within a single machine. If non-interference is important for an event source, the functionality can be split between two or more machines.

It is not readily possible to capture the concept of an interrupt with a machine as event handlers cannot be preempted, and 'terminating' active event handlers could result in inconsistent machine state. Therefore a hypothetical **deadline** event source that would terminate its event handler after a specified duration cannot be expressed. However a variant of the **periodic** event source could be constructed that would invoke one of a pair of event handlers depending on if the handler was being invoked before or after a specified deadline.

Machines and the timed event sources they contain cannot guarantee to be able to know 'calendar time' as this would depend on access to a Real-Time Clock (RTC) hardware resource. Local time relative an arbitrary but fixed point in time (such as the local processor boot time) can always be known.

3.2.4 Creating Machines

Machines cannot be explicitly *constructed* by application code, they must be requested from the implementation framework (discussed further in section 3.4). The exact syntax will depend on the implementation language but the general contract is always the same: a machine of a specified type is requested and at some point in the future a reference to the new machine is supplied. The requesting code cannot provide any parameters to influence the generation of the new machine, and no data can be supplied to the new machine for construction. The framework implementation has total discretion about the placement of the new machine in the platform.

Newly created machines are not given any information about their execution context, and are not supplied with a reference to their requesting machine. If an application needs a new machine to communicate with an existing machine then a reference to the existing machine must be sent to the new machine via a channel that the new machine contains. This architectural pattern can be seen in the example application (figure 3.2) where the **Flow Controller** machine requests a new **Level Sensor** machine and then sends a reference to itself to the newly created **Level Sensor** so that the new machine can communicate back to the **Flow Controller**.

There are no bounds on the duration of a machine request and the request is not

guaranteed to succeed. The way in which a machine determines that a request has failed is implementation defined, but the requesting machine must never acquire an invalid machine reference to a partially generated, malformed or non-existent machine. The possible failure modes for machine requests include hardware failure of local or remote devices, insufficient hardware resources that are required by the requested machine type, or any other possible implementation dependent failure mode.

Machines are not required to be immortal but application code cannot request the destruction of a machine, this is because a machine cannot *know* that it is safe to destroy the machine in question; another machine may still be using it. The runtime framework has the freedom to destroy machines on the condition that its destruction is not visible to the application code. This condition is not trivial and requires a framework to determine:

- That the moribund machine has no future spontaneous behaviour, or that any future spontaneous behaviour cannot interact with other machines.
- **and** no other machine owns a reference to the moribund machine that can be used for a synchronous or bidirectional interaction. Purely asynchronous and unidirectional communications could be tolerated to machines that do not actually exist.

3.2.4.1 Entry Point

One machine type is specifically designated as the entry point of the system. At the start of execution this machine is instantiated by the framework implementation and is the only application defined machine that exists when application code begins to execute within it. The **Flow Controller** machine is the *start* machine in figure 3.2 and this is denoted by its double outline.

3.2.5 Communications

Machines can only interact with each other via the *channels* defined in their machine types. Interaction can mean the exchange of data or establishing a synchronisation relationship. A channel contained in a machine can be used by any number of other machines, but each intermachine transaction has exactly two participants. Channels are a generalisation of the message box concept from other actor-oriented models: all

interaction is fundamentally based on asynchronous and unreliable message passing. There are many different *protocols* that can define a channel's behaviour and these are discussed later in this section. Instances of the machine type that defines a channel are always *reactive* to transactions initiated by other machines. Machines do not have to define or create connections in order to communicate.

An important concept within the application model is that every machine operates within its own *logical local area* and that there is not a valid 'global' view of the whole system at runtime. This ensures that the overheads of a machine implementation only need to be related to the magnitude of its logical local area and not the magnitude of the whole system. From the perspective of a machine, only its logical local area even exists at runtime and therefore only this area can be subject of interactions. However, a machine's logical local area is not fixed and can change over time. The consequence of this concept is that communications channels cannot support non-local addressing schemes such as:

- *broadcasting* — all nodes addressed
- *geocasting* — only nodes in a particular physical location are addressed [141].
- *multicasting* — addressing a group of nodes.

Broadcasting and multicasting within a machine's logical local area are compatible with the application model, but are not considered in any detail here as the same effect can be achieved within an application using only *unicast* interactions.

This section is divided into three major parts:

1. The qualities common to all channels are considered in section 3.2.5.1.
2. Section 3.2.5.2 discusses the protocol characteristics that are important for inter-machine communication.
3. Finally, drawing from combinations of the defined protocol characteristics, a collection of distinct channel types are named and defined in section 3.2.5.3

3.2.5.1 Characteristics of All Channels

A number of formalisms for computation demonstrate that only a single communications protocol is sufficient to express any system behaviour. These formalisms include

CSP [83] (with unbuffered synchronous channels), actor-model computation [81, 23] (with asynchronous, unbounded channels that can reorder messages), and Kahn Process Networks (KPN) [107] (with unbounded FIFO channels). In the context of a model of computation having a single communications channel protocol simplifies reasoning about systems and the addition of extra protocols does not improve the expressibility of the model; the models are already universal.However, for practical application development there are benefits to modelling multiple types of channel with differing properties. In this model all channels, regardless of the protocol that they implement, will have some features in common. These are:

single destination A specific channel is declared in exactly one machine type and cannot be shared across several machine types; all instances of a machine type will gain their own independent instances of the declared channels.

defined and known characteristics Channels have application-defined characteristics, discussed below, which are known by all accessor machines and implementation tooling.

static definition Channels must be defined in the machine type and they cannot be created at runtime. This is discussed at length in section 3.2.6.

multiple access A channel can be 'used' by any machine that owns a reference to the machine that contains the channel, and there is no model-imposed limit to the number of accessor machines for any given channel. This means that machines can have many-to-one communications relationships via a single channel.

message based Communication and other interactions are message oriented and not based on streams. Connections do not have to be established between machines to use a channel.

machine owned Channels do not exist in their own right but are best considered to be features of the machines that they are declared in. If the channel contains a buffer then it is stored in the machine that declares the channel.

event sources All channels are event sources in the machines that declare them, and bidirectional channels (discussed on page 88) are also event sources in the machines that use the channel.

undefined reliability and ordering The application model does not require that channels will guarantee delivery, provide error reports on failure, provide timeouts, prevent duplicates, or ensure messages arrive in the same order as they are sent. In addition, due to the distributed and decoupled nature of machines, even high-reliability implementations cannot ensure that the ordering of messages received from distinct senders can be guaranteed to be the same as the order in which they were sent. This is because the two (or more) sender machines have incomparable timing sources which cannot be used to order the message dispatch, and the sender machines will not necessarily be aware of each other's existence so cannot establish a logical send-ordering either. This does not mean that channel protocols cannot be reliable, but that the existence of a channel does not imply reliability.

Implementations that can target platforms with inherent unreliability may chose to assist application development by differentiating between reliable and unreliable implementations of channels, or by providing additional 'wrappers' for channels that can provide additional assurances for message delivery, ordered messages, and flow control. Additional assurances come at the expense of increased message latency and runtime complexity.

undefined blocking The application model does not define the local timing characteristics of channel operations. Implementations have the freedom to chose between the provision of *blocking* and *non-blocking* operations where appropriate. For example, if a channel is not yet ready to receive a new data item the sender event handler could be prevented from making any more progress (blocked) until the receiver is ready to accept, alternatively non-blocking schemes would allow the sender event handler to continue to execute even after the potentially not yet accepted datum was provided for sending. Blocking communications schemes are superficially very easy to program with as the natural flow of control is maintained, and blocking semantics are familiar as they are most common in popular programming languages for their primary networking and file system access features.

The machine model makes blocking communication less desirable as event handlers are non-preemptive, so if an event handler is blocked waiting for a channel operation then no other event handlers are able to become active either. Block-

ing channel operations can effectively stall entire machines from making progress. Even worse, blocking operations within a machine can allow distributed deadlocks to exist. Improperly synchronised machines with mutual communications dependencies can easily deadlock if they both wait for the other to send data.

3.2.5.2 Channel Protocol Characteristics

Even though a channel in a machine can have multiple other machines using it, each separate interaction is a point-to-point communication between a pair of machines. A quite extensive taxonomy of the protocols that can exist between two communicating processes has been explored by Simpson [191], and these can form the foundation of the useful protocols suggested by this application model, indeed many of the protocol names[1] used in [191] are carried into this application model. However, machines are not exactly alike general concurrent processes and channels are not modelled here as their own entities, so naturally there is some divergence from the originally observed taxonomy.

In the context of machine-oriented applications the interesting protocol properties are *buffering*, *destructivity*, *type* and *direction*.

buffering The size of the buffer in a channel if it is buffered at all. Buffering has implications for the memory requirements of the machines containing the channel, and for the expected usage patterns of the channel. Unbuffered channels require the receiver to be ready and waiting *before* the sender begins the interaction, otherwise the unbuffered datum is lost. Channels that are both unbuffered and non-destructive (see below) provide particularly strong instantaneous synchronisation between machines as the sender must be held up until the exact moment that the channel owner is handling the receive event.

destructivity In [191] both the destructivity of protocol reads and writes are considered, and they are defined as a write (send) operation that *cannot* be held up, or a read (receive) operation that *can* be held up. The concept of destructive receives (where the machine declaring the channel can be held up) is a poor match for a

[1] In [191] the name 'Channel' refers to a specific communications protocol, but in this thesis it is used as a name for a generic communications message passing concept. The specific meaning of 'Channel' defined in [191] is most similar to CSP's channels and therefore requires destructive reads which are not well suited to machine-oriented programs.

primarily event-driven architecture. The expected practice when a machine wants to wait for one of its channels to produce data is to 'do nothing' as the channel will trigger an event when a new datum has arrived (or interaction of another variety has happened). In a sense the machine has been held up as no progress has been made to process a datum that has not yet arrived, but the machine is not *actively* waiting and it can perform its other activities as normal. Conversely, non-destructive (always ready) reads are when the channel provides a persistent read buffer that can be consulted by application code repeatedly. This is always possible for unidirectional channels, and it is also trivial for applications to introduce their own non-blocking read functionality (a buffer) that is refreshed by the channel's "on received datum" event handler.

Destructivity of *send* operations to channels is very much applicable in a machine-oriented context. Channels that support destructive sends can always accept new data regardless of their current state, and this makes them easy to use and amenable to efficient implementations (as there is no need to consider flow control). In contrast, non-destructive channel protocols can only be sent to when the receiver is ready or has sufficient buffer for the new data item, and this substantially complicates both the send-side semantics and the implementation:

- Non-destructive channel sends require that the sender machine is able to know if the channel ready to accept another datum. This implies that the implementation must have an underlying bidirectional communications medium to allow the channel to signal its readiness to receive. The application layer should not need to be aware of the low level messages exchanged between a 'sender' machine and a 'channel owner' machine to indicate the readiness of a channel to accept another datum. However, there are two main approaches that can be used in the implementation layer to support non-destructive channel sends:

    1. Repeatedly send the datum until the channel issues a positive acknowledgement of receipt. This scheme has the advantages that it is has low memory requirements and only requires a simple state machine on the send-side of the channel. This scheme has low memory requirements as the receive-side of the channel is not required to keep a table of prospec-

tive senders; the senders will continue to attempt to send. The disadvantages of this scheme are equally clear: there can be a lot of redundant communication, possibly contributing to network congestion, and this scheme cannot ensure fairness or any other prioritisation for senders. The data accepted is just the data that happened to arrive at a moment when the channel had the capability to accept it.

2. Request permission to send from the channel. The channel replies with some indication of when or how the sender can transmit the datum. This can be done either by delaying an acknowledgement until a time when the sender is guaranteed to have its datum accepted, or by replying quickly with a description of when it will be acceptable to send the datum.

   This scheme has the advantages that the datum is only sent once so minimises unnecessary communications, and data can be accepted in order of any priority metric. The implementation of a non-destructive channel can chose amongst all waiting senders whenever there is spare capacity to receive more data. The most obvious use of this capability is to ensure some degree of fairness between all senders, such as by queueing send-requests and releasing them in FIFO order. More complex notions of fairness, such as a fixed share of capacity over a moving window of time, of more general *quality of service* can be supported though active acknowledgement schemes such as this. The very widely deployed Transmission Control Protocol [169] (TCP) is an example of a protocol that uses active acknowledgement between receiver and sender to provide flow control, guaranteed ordering and reliable delivery for a single connection, but it is not used to provide fairness or prioritisation between multiple connections.

   The disadvantages of an active acknowledgement scheme are greater complexity at both ends of the channel, and greater memory requirements at the receive end of the channel. The receive-end must maintain a record of all pending senders in order to be able to permit them to send when capacity becomes available, and the sender must have some record of how many reservations it has granted so that it does not permit too

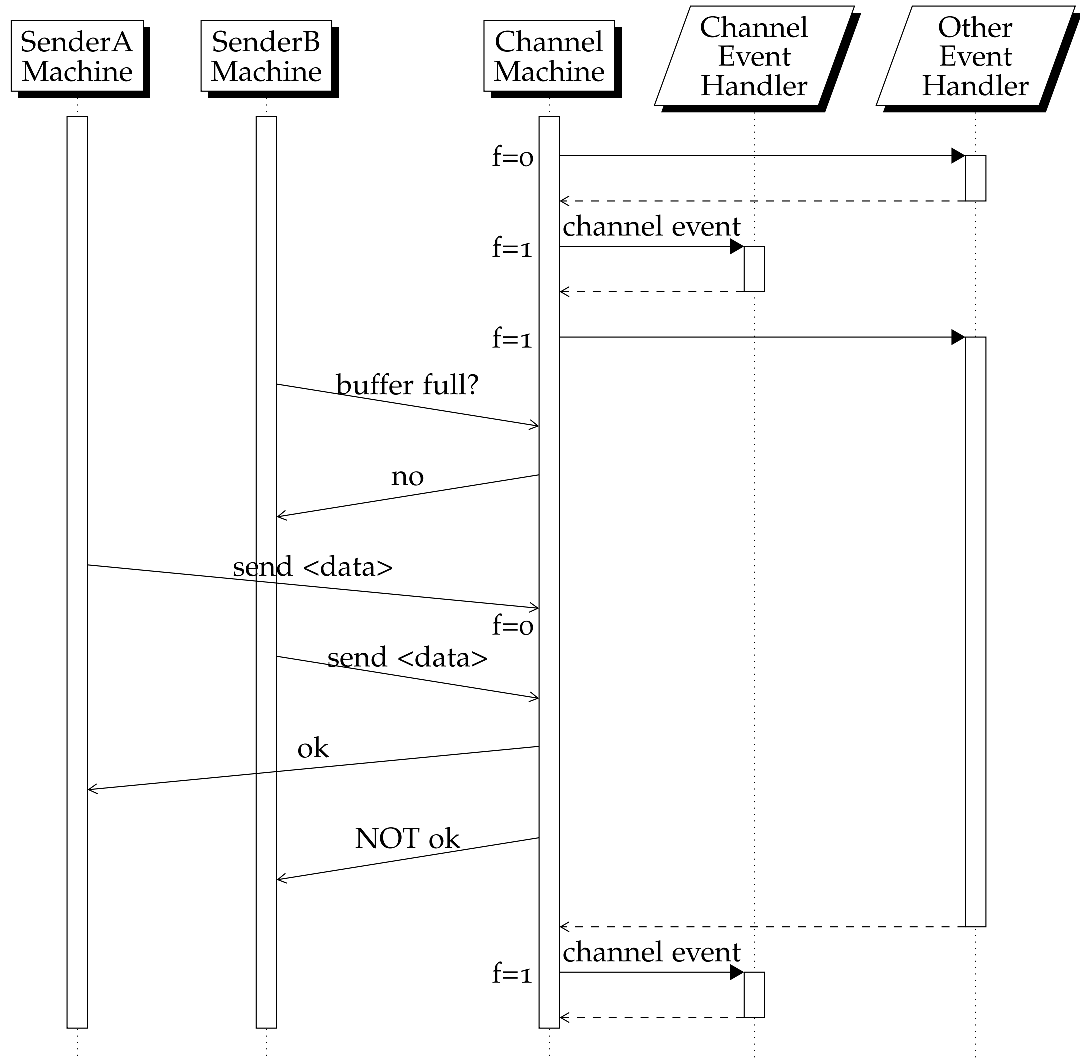


Figure 3.10: *A sequence diagram of the race condition that exists when non-destructive channels provide a naive 'buffer full?' query for application code to decide when to send. SenderB fails to send its datum even though it received an indication of available channel capacity. 'f' is the number of free buffer slots. 'f' is decremented on data receive and incremented when the channel has an event handled. SenderA and SenderB event handlers are omitted for simplicity.*

many senders to transmit simultaneously. This is further complicated if the delivery of the control messages is not guaranteed by the underlying communications medium.

- The second complication for non-destructive channel protocols is that the programming language implementing the model must be able to represent the semantics of a delayed send in some way. Non-destructive channels are the motivation for considering the blocking vs. non-blocking language semantics discussed previously. Implementations can chose between at least four approaches to providing non-destructive send operations:

1. Block the send machine from making progress until the datum has been accepted by the channel.
2. Provide an event when a non-destructive channel has become ready to send.
3. Provide a facility for application code to 'test' if the channel is ready to accept a datum.
4. Trigger an error condition such as an exception or error code if the send could not be accepted by the channel.

These approaches are not semantically identical and cannot be safely implemented at face value. Option 1 is always safe for the channel, but has the negative blocking consequences previously discussed. Approaches 2 and 3 are vulnerable to intermachine race conditions if they use an indication of the remote buffer's state to determine if it is the right time to send data (A "buffer full?" query). This scenario is illustrated in figure 3.10: SenderB checks with the 'Channel Machine' to determine if the buffer is full and even though the channel machine responds that its buffer is not full, SenderB still fails to successfully send data. The issue is that the response to the "can send?" query is only valid at the instant it was issued and is very likely to be invalid when it is received (an indeterminate amount of time later) by the querying machine. Other machines ('SenderA' in the example) are able to send a datum and use the available buffer space before the first machine ('SenderB') has its datum received by the machine with the channel.

Approach 4 is a variant of approach 1 as it must block for as long as it takes to either send the datum or determine that it cannot be sent, but it potentially shares a flaw with approaches 2 and 3 if it uses a remote "buffer full?" query internally. The potential flaws in approaches 2, 3 and 4 appear to suggest that non-destructive channels require blocking operations in application code, but it is possible to enable non-blocking programming styles with non-destructive channels. The key is that the "buffer full?" query must be reformulated as a "can send?" query that will reserve space in the buffer for the querying machine if the query is responded to affirmatively. With a buffer reservation scheme (the "active acknowledgement" scheme discussed on page 82) it is to design a non-blocking event-driven programming interface for application

code.

In the presence of faults, it is important that an implementation has a mechanism to release reserved buffer slots if the sending machine fails to send the data as promised. Notification of sender failure or timed reservations may be viable approaches but will further complicate an implementation. Where the expected number of sender failures over the lifetime of the system is much lower than a receiver's buffer capacity, it may be acceptable to simply accept permanent loss of buffer slots.

Figure 3.11 illustrates an example non-blocking exchange between a sender machine and a machine with a non-destructive channel that avoids race conditions between multiple sender machines. While this approach avoids blocking the sender machine, it does introduce considerable complexity into the application code. In the example an unrelated event handler begins the send transaction to the channel. The responses from the 'Channel Machine' are queued in the 'Sender Machine's event queue until the CanSend event handler is able to execute. The application must maintain a state machine for use by the CanSend event handler to decide what its action should be. Framework implementations would be able to ease application complexity by providing channels that implement this handshake internally. Implementations could also provide fixed, n-length buffering on the sender side for each channel that the machine communicates with. Sender-side buffering ensures that the first n data items sent could be 'accepted' from the application, and where fewer than n items are ever sent to the channel there would be no need to explicitly implement the handshaking logic.

A lesser characteristic of destructive channels is the way that they handle buffer overrun situations. When a new datum arrives but the buffer is already full a decision must be made about which datum to loose. Unless a priority can be determined for the messages, a *content neutral* policy must be used such as overwriting the least recently received (oldest) datum. This patten is sometimes [33, 29] referred to as *last-is-best* semantics, where only the most recent data are of concern to the receiver. Where the communication is about *events* that happened rather than *values*, each instance may be equally important and no policy for overwriting data is desirable. For these situations non-destructive channels or more sophisti-

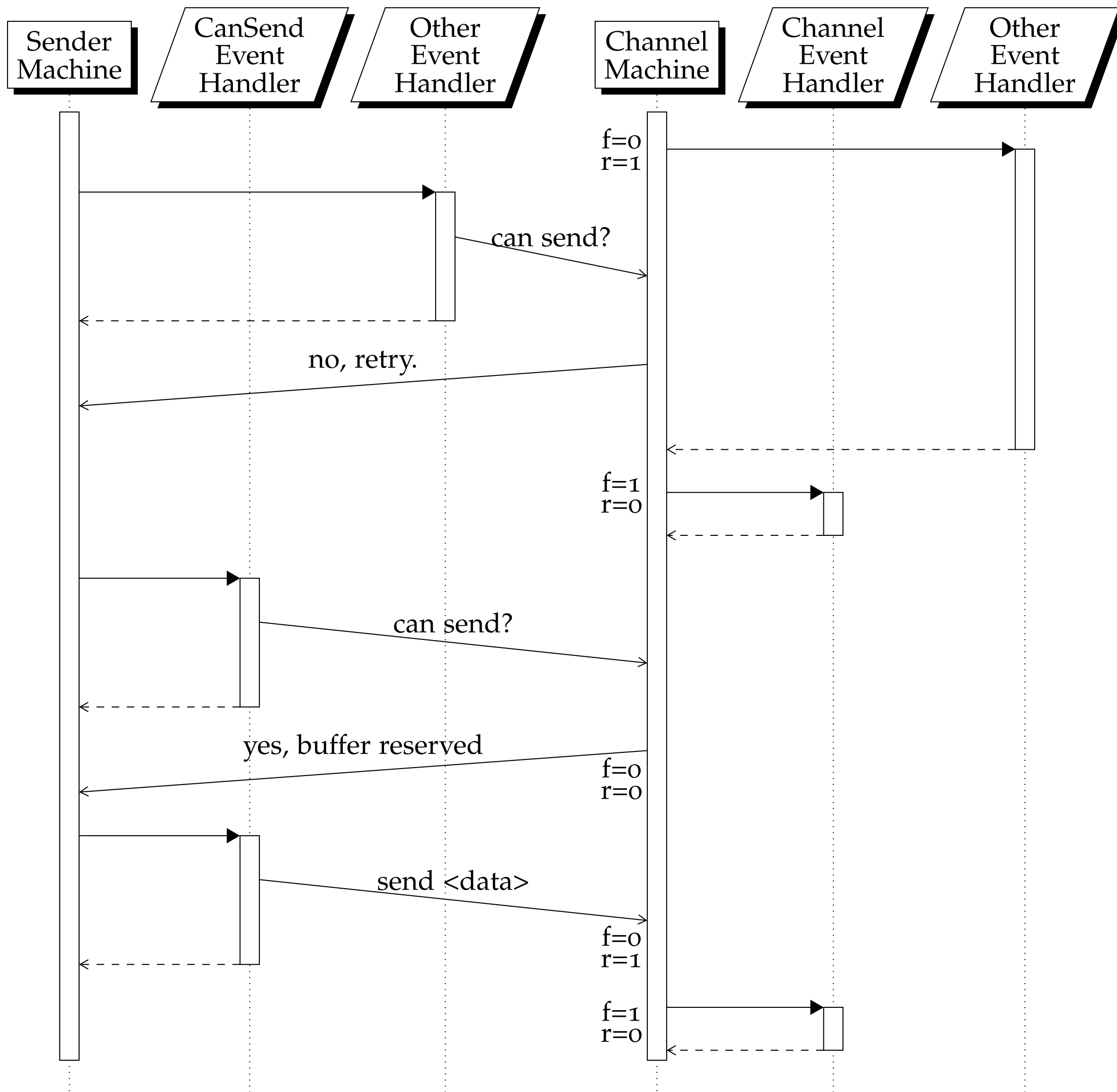


Figure 3.11: *A sequence diagram for a non-blocking send to a non-destructive channel. This sequence avoids race conditions between multiple sender machines. 'f' and 'r' represent the number of free buffer slots and the number of ready data items in the buffer, respectively.*

cated application level protocols are required to avoid unacceptable message loss. It is difficult to imagine the utility of *first-is-best* semantics, where the oldest data is retained preferentially, but it is an efficient scheme when the receive buffer is full as new messages are just discarded. First-is-best semantics are implicitly provided by implementations of User Datagram Protocol [167] (UDP) and raw IP [168] where incoming packets are simply dropped if the receive socket's buffers are already full.

type The *type* of datum that the channel is able to communicate. In [191] only two types of data are considered: ordinary data that has a value and a *void* type that has no value. This distinction enables void-typed channels to be defined which only provide synchronisation between the sender and receiver and do not transfer any data. This concept can be further generalised to allow channel definitions to differentiate between different types of ordinary data. The key principle is that a sender and receiver should have compatible expectations of what possible data values could be communicated via the channel, and additionally share expectations about an appropriate interpretation of the data. This ensures that if a channel is defined to communicate data of type T, application code can expect that only data compatible with its expectations of T typed data will arrive. Programming languages with *type systems* embed this concept into the semantics of the language [40], where it is expected that useful data is typed: it has some structure, pre-defined interpretation and importantly there are rules that define how it can be used meaningfully.

The specification of data types for channels is ultimately implementation dependent. Implementations in non-statically typed languages may also accept more loosely typed channels but unstructured or *raw* channels should be considered bad practice. This is for the same reasons that modern programming languages would discourage the use of an unstructured globally scoped array to store all of an application's state: It's highly prone to programmer error and there is little that compilers or other automatic tooling can do to help.

It's redundant to specify a type system in the application model and it would not be guaranteed to be a good match for any specific implementation language's type system. However, the general principle that implementations must follow is this:

> A channel of type T can accept a value of type U if and only if all values of type U can be assigned to a variable of type T.

Broadly, this allows channels to obey the value assignment rules of whatever type system an implementation language provides. This also enables languages with polymorphism to allow channels of polymorphic types.

direction The final differentiating characteristic for a channel is the 'direction' that it communicates data. In the application model the machine that defines the a channel instance is always the receiver and other users of the channel are the senders. More generally, machines that define a channel are *reactive* to transactions that are initiated by other machines to that channel. This general interpretation is necessary to accommodate void-typed protocols which do not meaningfully send or receive data.

Bidirectional communication can be established between a pair of machines trivially by ensuring that both machines define a channel for the other to communicate with and that the machine initiating the communications includes a reference to itself in the first message. This approach has the notable disadvantage that the non-initiating machine must have *prior* knowledge of the machine type of the originating machine, and specifically knowledge of the channel in the originating machine type that will be used to reply to the first machine.

This 'trivial' bidirectional communications construction has significantly limited expressiveness within the application model. A general purpose 'service' concept (where a machine of *any* type can request a service from a *server* machine) is inexpressible, as the server would be unable to reply to a *client* machine of a previously unknown type. This is not an issue for actor models that only have one channel per actor, or where the actor *is* the channel (such as Agha's model [5]). For single-channel actors there is no ambiguity about the possible return 'address' when a message requires a reply.

The expressiveness of machine-oriented applications can be substantially improved by allowing single channels to have a built-in return path, thereby becoming bidirectional channels. Such a channel must define its data types in both the query (towards the machine defining the channel) and response (towards the machine that issued the query) directions. This type of channel allows general services to

be provided by machines when when the client's machine type is unknown. The server does not need to know the machine type of the client as the channel defines both the return path and the response data types. The buffering and destructivity considerations largely identical to their unidirectional counterparts:

- Buffering is an important parameter for the machine that declares the channel as it cannot know how many client machines will use the channel, but return path buffering is less interesting. A machine can only get as many responses as it issues queries for, so the necessary return buffer size is only a consequence of the querying machine's design.
- Destructivity for outbound queries has the same implications as it does for unidirectional channels, but a client of a destructive bidirectional channel has no assurance that it will ever receive a response if the server is overburdened. All bidirectional channels must have destructive return paths as a server machine cannot be held up replying to a client, and cannot expend unbounded memory on return path send-buffers. This does not present an issue if the client machine guarantees buffer space for the responses of all in-flight queries to bidirectional channels. Buffer reservation for each in-flight query could be very expensive if the query issuing period (p_i) is substantially less than the total round trip time (p_r: the sum of outbound network latency, inbound network latency, queueing time in the server machine and processing time) from the server machine. The required response buffer capacity is

  $$\frac{p_r}{p_i} \cdot |R|$$

  where $|R|$ is the size of an instance of the channel's response type R. For variable-sized types, $|R|$ is maximum size of any instance in the subset of possible R typed responses.

3.2.5.3 Defined Channels

Various combinations of the the four important protocol characteristics (*buffering*, *destructivity*, *type* and *direction*) produce channels with recognisable semantics. The most important of the possible unidirectional channels are specified in table 3.1 and the bidirectional protocols are listed in table 3.2. For every *value* typed channel described an

instance can be defined in a machine type for any allowable type in the implementation language. Instances of different kinds of channels are illustrated in the example application in figure 3.2.

Bidirectional protocols differ from the unidirectional protocols because data can be exchanged in both directions, but they do not necessarily provide any stronger synchronisation between the machines. The unidirectional **rendezvous** channel (discussed on page 92) provides the strongest possible synchronisation between machines. The most important conceptual difference between uni- and bidirectional protocols is that the receiver not only receives data from the sender, it receives a tuple of a datum and a return path to the machine that issued the query. The return path is an *anonymous* **signal** channel (an overwriting single-length buffer). When the channel-defining machine (the server) sends its response to the return path it will queue an event as usual in the querying machine. Multiple responses by a server are not permitted by this model and can either be ignored or forbidden by implementations.

In the table of unidirectional protocols (table 3.1), there are only two protocols that do *not* have destructive reads. The **pool** and **constant** protocols uniquely provide channels that are always readable in their host machine. Events can still be raised when a **pool** is updated, but its purpose is to allow data to be updated from one machine to another without any temporal relationship being established.

As all channels are owned by their declaring machine type, **remote data fetch** channels nearly represent an inversion of the sender receiver relationship, but the non-declaring machine retains the role as transaction initiator and the channel owning machine still fulfils the server role. In the scenario where machine *a* will transfer data to machine *b*, the important difference between a **bounded buffer** and a **remote data fetch** is which machine defines the contract of the transfer. Using a unidirectional protocol, the data recipient (machine *b*) defines the contract but with a **remote data fetch** the data sender defines the contract. This enables a machine to be a data source for another machine of an unknown type.

Name	Buffer	Destructive?	Type	Purpose
Bounded Buffer[2]	n	R	value	Root of all non-overwriting channels
Slot[3]	1	R	value	Reliable single-slot buffer
Rendezvous[2]	0	R	value	Strong synchronisation with data transfer
Directional Handshake[2]	0	R	*void*	Strong synchronisation without data transfer
Bounded Stim Buffer[2]	n	R	*void*	Queue of action requests with backpressure
Overwriting Buffer[2]	n	RW	value	Root of all buffer overwriting channels
Pool[2]	1	W	value	Unsynchronised data updates (always readable)
Signal[2]	1	RW	value	Sender unsynchronised data updates
Overwriting Stim Buffer[2]	n	RW	*void*	Queue of action requests without backpressure
Stimulus[2]	1	RW	*void*	Action flag
Prod[2]	0	RW	*void*	Release an action only if it's already waiting
Constant[2]	1	–	value	Statically defined values

Table 3.1: *The variety of well defined unidirectional channel protocols.*

Name	Query Type	Response Type	Purpose
Remote Procedure Call[2]	value	value	Root of all bidirectional channels
Remote Data Fetch[2]	*void*	value	Request data
Remote Data Send[2]	value	*void*	**Bounded Buffer** with acknowledgement on processing
Remote Event Invocation[4]	*void*	*void*	**Bounded Stim Buffer** with acknowledgement on processing

Table 3.2: *The well defined bidirectional channel protocols.*

[2] Naming and semantics broadly consistent with [191]
[3] Referred to as a 'Channel' in [191]
[4] Referred to as a 'Remote Thread Invocation' in [191]

A **rendezvous** channel (and the dataless **directional handshake** variant) are not perfectly matched to machine modelled applications. A rendezvous more usually implies [24] that at least one of the client or server *waits* until the other is ready to proceed but this is not the expected mode of operation for a machine. If blocking operations are provided by the implementation then it is possible for the client to be delayed until the server machine is ready to receive the datum. However, it is only possible for the server to 'wait' for a client by refusing to service any other pending events. In situations where the implementation does not provide blocking channel operations the allusion to a rendezvous becomes even more tenuous as neither the client or the server are truly waiting for one another. Figure 3.12 shows how a notional non-blocking rendezvous would occur in a machine oriented context. As far as possible, implementations should hide the necessary synchronisation messaging from application code. In the figure the red numbers indicate important points in the non-blocking rendezvous' sequence:

1. The rendezvous begins for the sender machine when application code requests to rendezvous with another machine.

2. The machine defining the rendezvous channel (the receiver) rejects the request if there is another rendezvous in progress. If the rendezvous is accepted a notification is sent to the sender machine instructing it to wait. This message is sent without application intervention on the receiver. A "rendezvous is starting" event is queued in the receiver machine. No other rendezvous can be accepted by the channel-defining machine until this transaction has completed.

3. The receiver handles the "rendezvous starting" event, causing a "send now" message to be sent to the sender machine. This message is sent without application intervention.

4. The sender machine queues a "recipient ready to accept" event.

5. The sender machine sends the data to the recipient channel, and the rendezvous is now complete at the sender end. Sending the data *may* be completed without application intervention.

6. The receiver queues a "data received" event.

7. The receiver handles the "data received" event, concluding the rendezvous for the receiver. The receiver can now accept another rendezvous request.

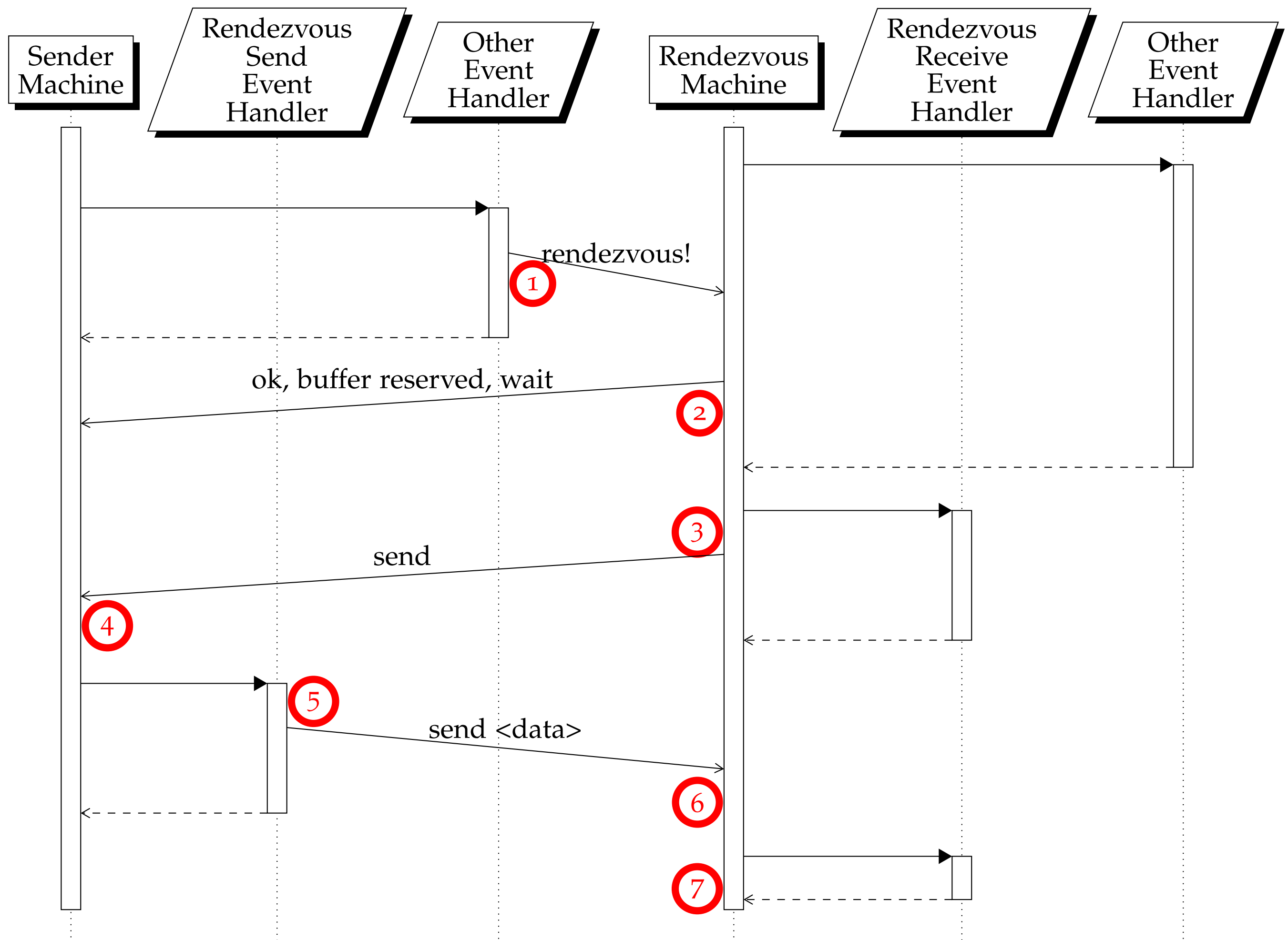


Figure 3.12: *A sequence of interactions that implements an event-driven, non-blocking rendezvous channel. The rendezvous machine's 'other event handler' is to illustrate the delaying effect of other activity within the machine.*

It can be seen in figure 3.12 that both the sender and the receiver will execute two event handlers during a rendezvous transaction, but here are no assurances that rendezvous event handlers in the sender machine will actually be live during the same physical time as the rendezvous event handlers in the receiver machine. Therefore this construction is a *logical* rendezvous as only the ordering of the event handlers is guaranteed. The two machines can be considered to be logically synchronised between instants (3) and (5) in figure 3.12: Only between these instants can either machine correctly assert that the rendezvous transaction must be active for both itself and the other machine, but application code associated with the rendezvous is only live for sections of this interval. This means that the receiver's synchronisation point is at (3) before the 'send' message is issued, as by the time that the datum is being handled at (7) the sender machine may (and is likely) to have finished it's rendezvous event handler. Likewise, the sender's synchronisation point is at (5) before the datum is sent, as before (5) has happened the sender application code cannot be sure that the rendezvous has begun on the receiver.

A non-blocking **rendezvous** is also only *logically* unbuffered as implementations will need to buffer the datum between its arrival to the receiver and the second event handler execution. It is logically unbuffered in the sense that both applications must have aligned to perform the transfer, and if the value is not used by the receiver machine it is lost.

3.2.5.4 Undefined Channels

Implementations of the application model would be expected to require applications to structure their communications in terms of named channel varieties rather than a more general 'omni-channel' concept. This is because for a given channel configuration the semantics can be well understood by humans and automated tools alike, whereas a general purpose configurable channel would facilitate combinations of protocol characteristics that may have poorly defined semantics, or may simply find an edge-case in the implementation.

The following are some examples of channel variations that are meaningless or have inconsistent semantics:

- A buffered, unidirectional protocol with non-destructive reads and writes. This general construction is described in [191] as a **constant** as the sender may be blocked forever without contributing any data to the channel, so it can be consid-

ered to be a read-only protocol. Only the data that was originally in the read-side buffer will ever be read. If this protocol is also void-valued then it contributes absolutely nothing to an application.

- **pool** protocols provide no synchronisation, so a void-valued **pool** would contribute nothing to an application.
- Bidirectional protocols with non-destructive server side reads generate a contraction of definitions. A 'read' from a bidirectional channel provides a *query* from a client that must be serviced once; they are single-use. However a protocol with non-destructive reads can always provide items to application code. These requirements cannot be reconciled. An application *could* buffer data items associated with previous queries.
- Generally, bidirectional protocols are best defined with destructive read operations, especially on the client side. It is meaningless for an application to non-destructively read the response of a query that not necessarily even been issued to the server.

3.2.6 Dynamic Synthesis and Communication of Application Elements

The characteristics of machine references are contrasted against the other elements of a machine oriented application in table 3.3. The *defined* column refers to the appearance of the specified element in the static definition of an application. *Tangible* elements are those that can in some way be manipulated in application code; if an element can be assigned to a variable at runtime it is considered to be tangible. Where an item has been marked with an asterisk(*) in table 3.3 this indicates that the model is consistent with either possibility so implementations are not constrained in their choice to support or restrict that possibility.

The most important properties of elements in a machine application are described by the capability of application code (contained in event handlers) to either synthesise or communicate that element. These capabilities will now be considered in some depth as they make significant contributions to the rationale of the application model.

Entity	Synthesisable?	Communicable?	Defined?	Tangible?	Origin	Enables
machine types	✗	✗	✓	✗	definition of application	static application structure
machine type names	✗	✓	✓	✓	definition of machine type	request a new machine of named type
machines	✗	✗	✗	✗	implementation defined	execution context and data storage
machine references	✗	✓	✗	✓	generated by implementation framework	communication with referenced machine
channels	✗	✗	✓	✗	definition of machine type	static communications structure
channels names	✗	✗	✓	✗	definition of machine type	communication with named channel
event handler	✓	✗	✓	✓*	machine type definition or dynamic generation	any behaviour of a machine
mutable data	✓	✗	✓*	✓	created and updated by event handlers	machine local state
immutable data	✓	✓	✓*	✓	created by event handlers	intermachine arbitrary data sharing

Table 3.3: *A summary comparison of the various elements in a machine oriented application, including the ability for an element to be communicated and the purpose of the element. *Indicates an implementation defined possibility consistent with the application model.*

3.2.6.1 The Synthesis of Application Elements

Synthesisable elements of an application are able to have new and usable instances generated by event handler code. For example, event handlers are able to generate new instances of integers, lists or any implementation permitted data structure. If the implementation language and runtime supports it then even new event handlers can be generated dynamically. However most structural elements of a machine application cannot be synthesised by any event handler:

machines cannot be generated by application code because the nature of their execution context and memory isolation is necessarily framework-implementation specific and likely platform specific too. Platform independence for applications is one of the primary goals of the machine abstract architecture. Applications must of course be *implemented* in a programming language in order to be useful but it is incongruous for an application to contain code to construct its own machines as this creates framework-implementation specific dependencies.

machine references to unseen machines cannot be generated reliably by any application code no matter how the code is constructed. There are two factors that cause machine references to be unsynthesisable: they are opaque and a machine does not (by definition) have enough information about the global state of a system to generate a reference to another machine. The opacity of a machine reference means that the internal structure of the reference is framework-implementation defined and unavailable to the application code. Only the type of a machine and that the machine exists[5] can be determined from its reference. As above, an application would lose portability if it contain code to generate machine references as it would generate references with an implementation specific structure and would either have to guess about the (non) existence of other machines, or use some other implementation specific mechanism to locate them.

The non-synthesisability of machine references is not a rule that must be enforced by language frameworks, but a framework constructed in a permissive language such as C might chose to provide some protection against accidental corruption or 'light-hearted' tampering. Implementations in safe or managed programming

[5] Even the machine's existence is not *guaranteed* – machines can be destroyed on the condition that it cannot affect application consistency.

languages such as Java or C# could make use of the built-in language references which cannot be manipulated by application code.

channels do not make sense to be synthesised dynamically due to the way in which channels are defined. Channels are part of the static definition of a machine so are fixed in number and have defined characteristics (see 3.3.3). Channels can only exist in the context of a machine type, and are only useful if some other machine is capable of communicating with it.

Applications cannot make assumptions about how channels will be implemented or even if they will have a 'data' representation at runtime. Implementations may chose to map a machine type's channels to real hardware communications resources, and in this case there is absolutely no possibility of generating a new channel at runtime; the channels are defined by and fixed into the hardware.

channel names are used by application code to reference a channel defined by a machine type. To be able to successfully communicate *to* a channel a machine must both have the name and know the characteristics of the channel. The definition of a channel in a machine type is an indication that it has a specific meaning or purpose to the application as a whole, so even if some application code could synthesise a valid channel name dynamically there would be no meaning to communicating with a previously unseen channel. For this reason channel names are not considered to be synthesisable dynamically. It is possible to construct an implementation in which a machine could dynamically enumerate the channels that a given machine type defines, but this does not help a machine to understand *why* a channel exists, and therefore does not enable a machine to communicate sensibly with channels it was not explicitly coded to interact with. The ability to enumerate channels of a machine type would not constitute dynamic *synthesis* per se as the channel names must already exist in some form in order to be enumerated.

The channels defined by a machine type form the externally visible contract of that type to all other machine types. This makes them analogous to the members of classes or structures in existing programming languages, where it is common for the elements of a class or structure to only be accessible by identifiers that are statically encoded into the application. For example, the members of a Java class can only be accessed via their identifiers which must be statically encoded into

the source code. Without resorting to reflection, new identifiers cannot be created at runtime to a allow a method access to previously inaccessible class members. In the C language access to the members of `struct` types is similarly limited; The members of a C `struct` can only be accessed by using their identifiers or by reinterpreting the structure as another unstructured data type. While Python [176] makes it far easier to access the members of a class dynamically (though the use of the `getattr` and `setattr` functions [177]) the primary syntax of the language still requires an identifier to access a class member.

machine types *could* be synthesisable at runtime but it presents such severe challenges for both implementation and analysis that it is essentially infeasible. The issues include:

- If the behaviour of a new machine type is novel at runtime (such as if the code were loaded from a network or storage device), then there would be no way for implementation tools to determine the possible structure of the application as the application definition has been effectively hidden from the compile-time tools. No meaningful analysis of an application beyond the statically defined machine types is possible.
- New code in a new machine type must somehow be communicated to other platform processors. This means that an implementation must have a mechanism to determine where machines of the new type could be allocated dynamically, and to ensure that those remote processors have access to the code.
- New machine types could only be synthesised at runtime if the implementation supports some form of runtime subtyping of an existing machine type. New channels defined in the dynamically created machine type could never be accessed by any existing machine type as they would not have the non-synthesisable channel names required to communicate with it. Furthermore, existing machines would not 'know' the purpose of the new channels. Assuming there is no dynamically loaded code in the new machine type, the motivation for such an approach is unclear as dynamically constructing a new machine subtype cannot achieve any outcome that was not already achievable. A dynamically created machine subtype is equivalent to a statically defined machine type that has behaviour that can change to the potential sub-

type's behaviour under the same circumstances as the subtype would have been created.

- As with the dynamic synthesis of channels, applications cannot assume that machine types will even exist as data at runtime. If a machine type has been mapped by an implementation to a specific function accelerator in the platform then there is no possibility of 'extending' this type; the behaviour is fixed by the hardware.

For the primary reason that application analysis is all but destroyed by runtime machine type synthesis, it will not be considered to be allowable in this thesis.

Machine type names are symbolic references to machine types. They are not synthesisable by application code for the same reasons that references to machine instances are unsynthesisable: to do so would damage portability and scalability of an application. Implementations have the freedom to partition an application across the available platform in such a way that not all machine types are available to all processors. No mechanism is defined that would allow application code to determine which machine types are available on the current or any other processor, therefore application code cannot guarantee to create valid machine type names.

3.2.6.2 The Communication of Application Elements

Just as there are restrictions on the synthesis of application elements, only a subset of elements can be communicated via channels. In general an application element can only be communicated if the communication is conceptually compatible with the machine model. There are two criteria to determine if a communication is conceptually compatible:

1. After the communication has concluded, are the machines still isolated?

2. Can the element be copied?

Isolation After Communication

The most important aspect of a machine is its isolation from all other machines, and this isolation can only be maintained as long as no communication implies that machines must share the same data afterwards. This means that mutable data structures cannot

be communicated between machines in such a way that modification of the data structure on one machine is visible from any other machine. The criterion that isolation is maintained does not imply any particular programming language design; any scheme that preserves intermachine isolation is acceptable, including:

explicitly copy mutable data to be communicated. Copying is implied if the underlying implementation would use a message-passing technique (such as a network) to communicate with the recipient machine. Copying the data is acceptable as even though the copy will be mutable, no modifications to the copy will be visible to the originating machine nor would they be expected by an application programmer. If communication of mutable data types is permitted by an implementation this may introduce a source of considerable confusion unless it is very clear to the application programmer that copies are always made of communicated data.

only allow immutable data types to be communicated. Immutable data types are fixed after their creation and have very desirable properties for machine applications: Flexibility of implementation and clarity for the programmer. Immutable data types enable implementations to have complete freedom to choose their low-level communications mechanism. If the recipient machine is hosted by a processor that has access to the same memory regions as the sending machine then it may be unnecessary to copy any data at all.

The implications of communicating immutable data with other machines are clearer too: the original data cannot be modified by the recipient as it cannot be modified by any machine. With immutable data types it does not matter if a programmer misunderstands what mechanism enabled a datum to be transferred from one machine to another as all mechanisms will have functionally identical consequences. However, immutable data types may be more difficult to work with as they cannot be incrementally updated after their construction. This may imply more redundant copying of data, and therefore greater overheads, than if mutable data structures were used instead.

Copying Application Elements

The second criterion for the communication of an application element is whether or not it can be copied meaningfully. This is because communication via channels fundamen-

tally requires that the data to be sent is copied as the sending machine is permitted to retain the data it sends. Application elements are only considered to be copiable if a copy of the element is indistinguishable from the original. This results in a very similar consideration as to whether or not an element can be dynamically synthesised by application code.

Immutable data is trivially copyable and mutable data structures could be considered copyable in most contexts. However, Machines (which are the combination of data and an execution context) cannot be meaningfully copied as their execution context is effectively a 'reference' to the capability of some hardware to perform actions, and clearly this capability of the hardware cannot be copied at runtime. If the data component of a machine were to be copied but a new execution context is assigned then this is effectively a new (but somehow pre-initialised) instance of the same machine type, and so would have a new, unique machine reference; the machine has not truly been *copied*. In contrast, machine references can be freely copied as they have no implicit dependencies on any hardware resources, only on the existence of the machine to which they refer.

In the case of channels, it is difficult to define what it would might mean to create a copy of a channel in order to communicate it to a remote machine: Channels only exist as features of a machine type and cannot exist in isolation and they may not have any data representation that would be possible to copy at runtime. As channels are conceptually non-copyable, they are certainly unable to be sent as data items between machines.

Summary of Application Element Communicability

In principle an application element *could* be communicated between machines via channels if it is compatible with the model of machines as isolated entities, as described in the previous criteria. However, for reasons of practicality it is useful to further restrict what types of data can be communicated, resulting in only three kinds of application element that can be communicated: machine type names, machine references, and immutable data types. The disallowed elements, and the rationale for their restrictions, are as follows:

mutable data cannot be communicated between machines as it implies that data will be shared by two or more machines after the communication.

machines and channels are not conceptually copyable. Any attempt to copy a machine or channel will not yield a result that is identical to the original.

event handlers are plausibly communicable as both necessary criteria can be satisfied, but the consequences of allowing event handlers to be communicated between machines are quite undesirable:

- As event handlers are the program code of a machine, the exchange of event handlers is only useful if the code is executable in the recipient machine. This is achievable either by ensuring that the event handler code is portable across all processor architectures in the system's platform, or by ensuring all processors in the platform have compatible instruction set architectures. Neither of these possibilities are desirable as architecture independent representations of event handler code (such as a programming language source code or virtual machine bytecode) will be substantially less efficient than native processor code, and requiring the homogeneity of a target platform is neither realistic nor an acceptable constraint for an application to impose.

  As applications cannot be permitted to dictate the characteristics of an implementation platform, the use of an architecture neutral event handler representation would become necessary to enable intermachine event handler communication. For very restricted platform processor architectures the overhead of runtime compilation or interpretation of event handler code is likely to be unacceptable too, leading to the conclusion that event handlers cannot be communicated.

- Event handlers are the only mechanism for a machine to update its internal state, and event handler code can only update the state of the machine that it is defined within. For event handler code to be usefully sent to a machine of a different type than the sender, the event handler must depend on access to the machine's internal state as this cannot be expected to have the same structure across machines of different types. In addition, a received event handler cannot expect to retain access to the state of the sender machine as this would break the isolation of the machines.

  Event handlers that *do* access machine state could be exchanged between machines of the same type if the implementation can guarantee that the machine

state will have the same structure across all instances of the same machine type, and the event handler code is expressed in such a way that it only ever attempts to access the 'local' machine's state.

- Program code encapsulated in procedures, functions, methods or event handlers are often not *tangible* or "first-class citizens" [38] of a programming language. This means that they cannot be passed as arguments, assigned to variables or returned from functions. Allowing event handlers to be communicated would require that an implementation uses a programming language where code is indeed treated as a first-class citizen, therefore substantially limiting the range of available implementation languages.

Overall, supporting communicable event handlers is highly problematic and provides little if any extra expressiveness to the machine-oriented programming model. For this reason the communication of event handlers is not allowed by the application model.

channel names also satisfy both necessary criteria for communicable elements but cannot be communicated as this would enable runtime communications patterns that were not apparent in the definition of an application. So long as channel names are neither synthesisable nor communicable, they must be statically encoded into application code to be used. Therefore forbidding the communication of channel names ensures it is always possible to extract the possible communications structure from the machine type definitions.

In contrast, the following three application elements *can* be communicated between machines:

immutable data can always be communicated between machines and therefore this is the root of all *guaranteed* communicable data. This is because immutable data communication cannot interfere with the spatial or temporal isolation of machines. Additionally, immutable data communication has less 'surface area' for misconceptions about the eventual behaviour of an application.

machine references are immutable and therefore can always be communicated. The exchange of machine references is fundamental to the application model as it allows machines to communicate with other machines that they did not request

themselves. Without the exchange of machine references, the runtime structure of applications would be limited to *trees* of machine instances, where machines can only initiate communication with their direct children machines. In the worst case, propagating data between two 'leaf' machines in such an application would require that the data flow all the way to the root of the tree and back, and there would be no way for each machine in the path to the root to notify its parent machine of it's desire to send data. Each parent would have to periodically poll its children for data.

Allowing machine references to be communicated enables an application to elaborate arbitrarily complex graphs of machines that are limited only by the resources available in the platform.

machine type names The *names* of machine types can be transferred as they are immutable. As machine type names are required to request machines, this enables the implementation of 'construction proxy' patterns within an application. A construction proxy machine can request machines of previously unknown types on behalf of other machines. In turn this enables a *machine pool* pattern to be constructed within an application, where a machine acts as a manager for a collection of machines with some common characteristics. Machine pools can be used to manage many-to-many relationships between machines that require access to certain resources and machines that own the resources. Machine pools can also provide similar utility to general purpose 'thread pools'. In this case a set of 'worker' machines are pooled and can be assigned work as needed. The main advantage conferred by this technique is that resource consumption can be more easily bounded than if a new set of machines are requested for each unit of work. Worker pooling may also increase the efficiency of an application as it avoids unnecessary instantiation (and eventual destruction) of machines, but this cannot be assured as the overall performance of a machine pool will depend both on its own characteristics and on the usage patterns if other application code issuing work. As with thread pools (such as the cached and fixed thread pool implementations in Java's `java.util.concurrent` [149, 148] package), worker pools could be designed with fixed numbers of worker machines, or with the ability to grow and shrink according to the prevailing workload.

Synthesisable?	Communicable?	Interpretation
✗	✗	intangible entities represented by identifiers or types
✗	✓	implementation supplied values for model-abstracted entities
✓	✗	any mutable data or code
✓	✓	immutable data

Table 3.4: *Patterns in table 3.3 lead to four interpretations for the combinations of synthesisability and communicability of application elements.*

> A *Machine factory* pattern, analogous to the conventional factory patten in object oriented programming but where it is a machine for creating other machines, can also be expressed in a machine oriented application but this is not predicated on the ability to communicate machine type names.

Table 3.4 summaries the four combinations of synthesisability and communicability for application elements and provides an interpretation of their meaning within a programming language.

These conditions result in the structure of interaction between the types of machine being statically encoded into the application, but not the multiplicity of each machine. The runtime structure of an application can be said to be *procedurally generated*[6] by the application code itself. This is substantially more expressive than a static description of an application's structure: all static application structures can be generated procedurally but the opposite is not true; static structure cannot make use of information available only at runtime. The application in figure 3.2 (page 63) is a basic example of runtime elaboration, where the runtime structure does not exactly match the static structure implied by the machine type definitions. A much more complex example of dynamic application elaboration is considered later in section 5.3.4.1.

This application model represents a tradeoff between an entirely statically elaborated model (which cannot be considered to be a true actor model) and a fully dynamic model where no application structure is guaranteed to be extractable.

[6]This term is more usually associated with computer graphics and specifically the runtime generation of textures or geometry according to an algorithm, as opposed to loading the asset from a disk or network.

3.2.7 Resource Usage

Platform specific behaviour in an application requires interacting with *resources* (see 3.3.2) that are defined to be available to specific processors in the target platform. In the application model resources are just abstract requirements that form a resource requirement set for each event handler in each machine type. Only the requirement that a resource is exclusively available to a particular event handler is captured. All other concerns, such as the mechanism for runtime interaction with a resource are implementation defined. The resources are considered to be used at an event handler granularity as these are the fragments of program code that form the aggregate behaviour of a machine.

It is not expected that event handlers or machine types will explicitly list their resource requirements but the requirements are deduced from the defined behaviour. For example, the requirement for a local floating point unit (FPU) could be inferred for a machine type if an event handler were to manipulate any floating point types. Some resources (such as external actuators) cannot service two concurrent accesses and therefore make most sense when they are required and exclusively accessed at a machine-level of granularity. This can be accommodated by 'coarsening' the requirement on a resource from an event handler to all event handlers defined by a machine. It is valid to over-provision resources for an event handler, but the opposite is not true: An event handler obviously cannot complete successfully if a required resource is not made available to it.

Regardless of the implementation defined representation of the set of resource requirements for an event handler or a machine, they must be comparable to the sets of available resources in the target platform in order that a legal mapping from machines to processors can be found.

3.3 Platform Model

The *Platform* is an abstract representation of the hardware architecture in a system. Its purpose is to allow decisions to be made about the allocation of machines across the hardware target at design, compile and runtime. Only the essential structural and behavioural characteristics of the target are captured by platform instances; it is not a hardware description language and is not intended to provide enough detail to reproduce

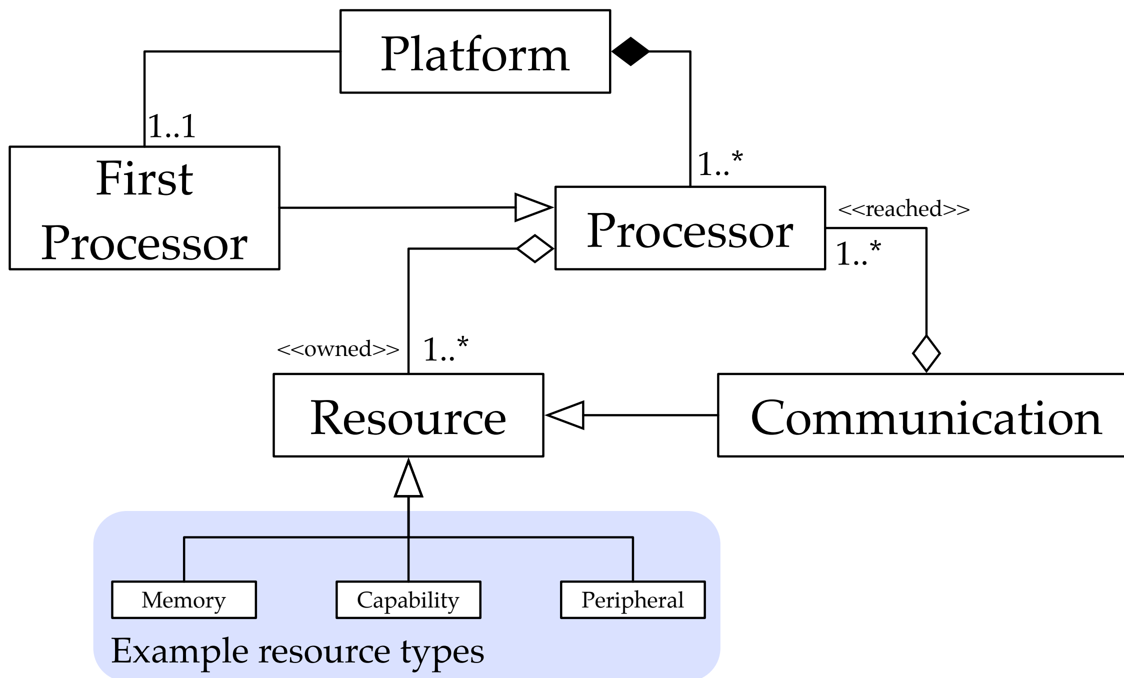


Figure 3.13: *An entity-relationship diagram for the platform model. Note that processors must have at least one* communications *resources but can have any number of other resources.*

the platform exactly. A entity-relationship diagram of the platform model is provided in figure 3.13.

At the top level, a platform is defined as a set of processors (section 3.3.1), each of which has a set of resources (section 3.3.2). Processor resources can represent memories, peripherals, or communications interfaces (section 3.3.3). In turn each communications resource has a set of processors that are reachable via that interface. Finally, a platform also has a defined *first processor* which is the first processor to begin its execution upon system startup. The first processor is where the start machine (see section 3.2.4.1) will be created and begin its execution of the application.

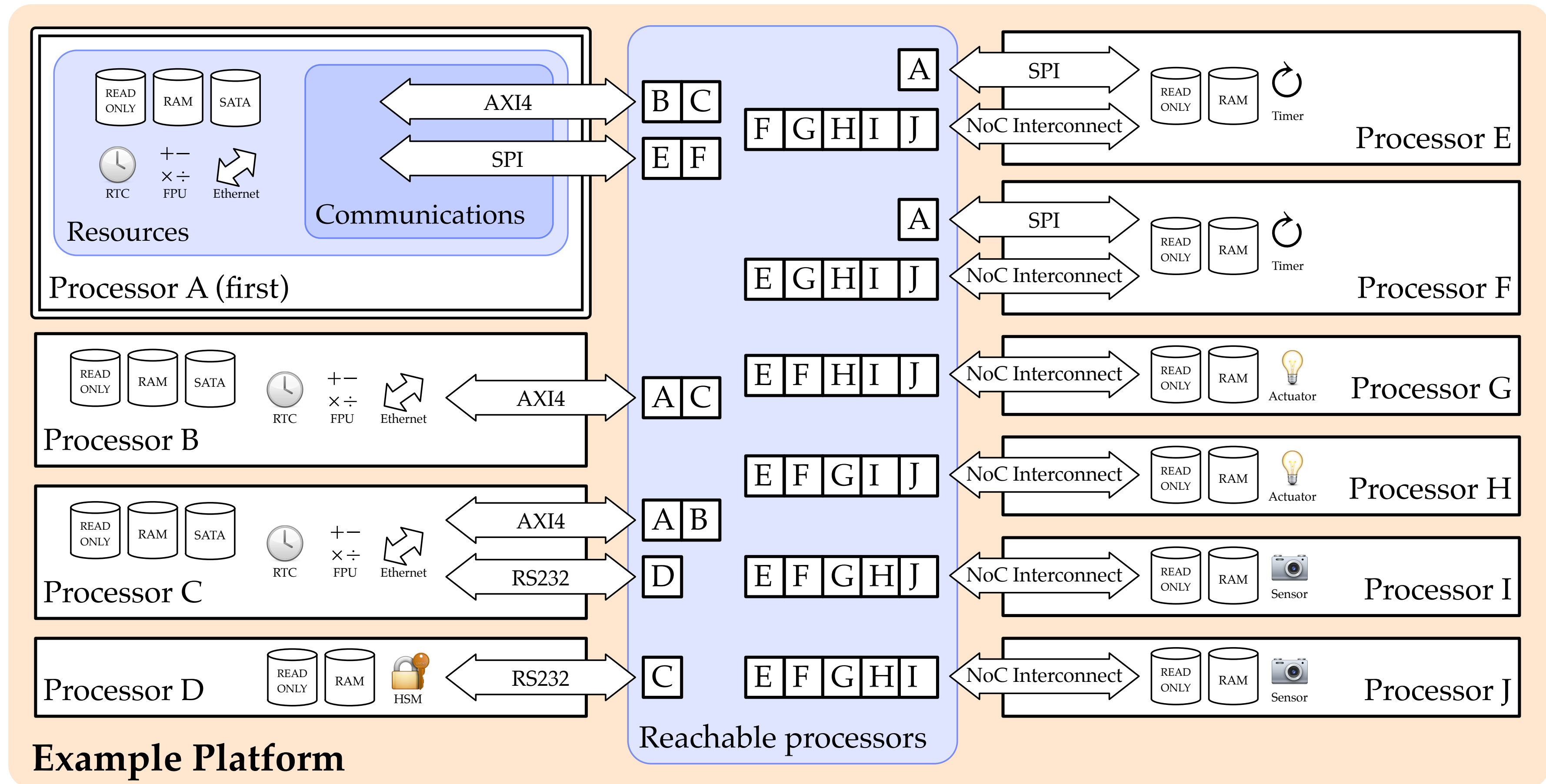


Figure 3.14: *An example platform illustrating the details captured by the model. In this example there are three powerful computing units connected by an AXI4 [227] bus, a 3×2 2D mesh network and a programmable hardware security module.*

In figure 3.14 an example platform is provided to illustrate that non-uniform topologies and heterogeneous architectures can be captured. The shaded regions in the figure indicate what details are represented by an instance of a platform model. The connectivity graph is the most important aspect of a platform and this is what informs decisions about how to allocate an application's machines at runtime. The platform in figure 3.14 has ten processors, three of which (A, B & C) are connected to a conventional AXI4 bus, one (D) is a programmable security coprocessor to (C), and the remaining six are part of a 3×2 mesh network. It is difficult to get an intuition for the layout of such a system just from the reachability sets of each of the processor's communications resources. Figure 3.16 shows the physical layout of this example system and figure 3.15 shows the logical *Processor Connectivity Graph* (PCG) that is encoded but opaque in the detailed figure.

A platform's PCG is a directed graph where there is a vertex for each processor and there is a directed edge between every pair of processors (p, q) from p to q if and only if processor q is a member of the reachability set of any of processor p's set of communications resources. This definition allows for the possibility platforms where there are unidirectional communications paths. Implementations may be somewhat restricted if there is no data return path between a pair of communicating processors, but these situations can be considered to occur in real systems: such as with very low cost infrared or radio frequency messaging, and when receiving broadcast data such as GPS coordinates.

A platform must have a *connected* PCG; there must exist some communications path from every processor to every other processor in at least one direction. It does not make sense for a platform to have disconnected regions of processors as this leaves no opportunity for them to operate as a single system. If the PCG disconnected then it could be considered to represent a platform for each connected component of the PCG.

3.3.1 Processors

Platform *Processors* represent the distinct concurrent processes that will execute the application at runtime. As they are only logical processors they may have a straightforward 1-1 mapping to physical hardware processors, or represent a processes in an operating system, or even represent the threads of a process.

The most important quality of a platform processor is that the resources that are allocated to it cannot ever be *observed* to have been modified by another processor in the

Figure 3.15: *The Processor Connectivity Graph (PCG) of the platform shown in figure 3.14.*

Figure 3.16: *The physical connectivity structure of the platform shown in figure 3.14.*

system; the processor appears to have exclusive access to its resources. A physical processor with several resources can be modelled equivalently as any number of platform processors as long as each of the resources are allocated exclusively to a single platform processor.

Platform processors are not required to have any mutual timing or synchronisation relationship to one another. The application model is equally applicable to platforms with a shared clock and those with no reliable common timing source. From this point, unless specified otherwise, the word 'processor' refers to a platform processor described in a platform model instance.

3.3.2 Resources

Processor *resources* are defined as any platform specific hardware or service that allows interference between any concurrent accesses to that resource. This interference can occur in either spatial or temporal domains:

spatial interference is where some data or state of a hardware device (such as a communications controller) has been altered by another concurrent process; the 'space' of one process has been affected by another. This has important implications for the safety and security of a system at runtime as processes or data can be unexpectedly corrupted or overwritten, hardware devices can appear to spontaneously enter unexpected states and data that must be kept private to a process could be read.

temporal interference is where the timing characteristics of a process are affected by the execution of another process. This is a very common mode of interference in the situation where processes are actually tasks or threads that share the same execution unit. Temporal interference is particularly important in real-time systems where the timing aspect of a system's behaviour is as important as the functional behaviour.

The specific structural constraints that the platform model imposes are intended to prevent spatial and temporal interference from being observable at the application level. Spatial interference can be 'trivially' addressed by enforcing exclusive allocation of resources but temporal interference is far more pernicious; it can creep in from multiple sources including those which are not usually hazardous to share, such as separate re-

gions of the same memory bank. Shared memory can introduce timing interference though the limited bandwidth of the memory controller and interconnects, but also via unpredictable interactions in memory caches [120]. Interference from implicitly shared hardware devices cannot be addressed by this resource abstraction. If it is critical that *all* temporal interference is eliminated from a system then all shared hardware facilities must be divided in some way that isolates temporal interference. For a network on chip this could mean the use of TDMA for the interconnect [72], or for an execution unit this could mean the use of an enhanced ISA that provides timing guarantees [36]. Once guarantees can be provided about temporal non-interference then implicitly shared hardware can be successfully abstracted in the platform model as multiple separate resources that can be safely used concurrently.

In the opposite case where a hardware resource can be accessed concurrently *without* interference, it need not be modelled as a resource on the condition that it can support as many concurrent accesses as there are platform processors that might access it. A RAM with as many ports as processors that can access it, and where each processor uses a non-overlapping region of address space is an example of a hardware resource that would not require any explicit mention in a platform model; this type of device could not introduce any interference in concurrently executing software.

Examples of typical resources that could be allocated to a processor include:

- Memory devices (or memory regions) for code, data, and ROM.
- Non-volatile storage such as flash memories, battery backed RAM and hard disk drives.
- External network interfaces and communications devices that are not used to communicate with other modelled processors.
- Peripherals such as sensors, actuators, timers, etc. Peripherals can also model platform capabilities which cannot be used concurrently, such as an authentication token for a remote server or a local hardware accelerator.
- Communications interfaces used to communicate with other platform processors. These are discussed in more detail in the next section (3.3.3).

In many cases peripherals will be accessible to multiple physical processors in the hardware, such as when peripherals are located on a shared interconnect, or with on-

chip peripherals in multicore microcontrollers. In this situation the resource must still be assigned to a single platform processor. If at runtime machines require access to resources on remote processors then this can be arranged via an explicit proxy pattern in the application layer. A sufficiently sophisticated implementation may chose to virtualise resources in order to allow a particular resource to appear to be available on several logical processors, and this strategy has been demonstrated in the implementation of embedded processor hypervisors [179]. Virtualising hardware peripherals does not conflict with the platform model on the condition that the illusion of exclusive access to local resources is maintained for each platform processor.

Hardware resources that are intended for synchronisation and inter-processor communication do not make sense to be allocated exclusively to a single processor; they only have utility in the context of multiple access from processors. Resources of this class which include hardware mailboxes, locks, FIFOs and shared memory, should not be modelled as ordinary local processor resources but are more appropriately represented as a communications resource.

Non-communications resources are only modelled to allow valid mappings from the application to the platform. All details of what the resources actually *are* and how they work is the responsibility of the implementation.

From a programming perspective, processor time (and the ability to execute instructions), memory and the ability to communicate with other machines are considered *abstracted* resources and do not need to be explicitly requested by an application. All other resources are considered *non-abstracted* and their use must be clear from the application code, either by explicit declaration or reliable analysis technique.

Interrupts from non-abstracted resources are not modelled explicitly but it an implementation could provide a device-specific event source for machines that use the resource.

3.3.3 Communications Resources

Platform *Communications* resources allow processors to exchange messages with other processors. They can represent any communication interface or implementation mechanism that is able to reach another processor represented in the platform. A device is considered to be a communications resource based upon its purpose as a message passing mechanism rather than an intrinsic quality of the device.

Processors A,B & C in figure 3.14 are shown to include ethernet resources but these are not modelled as communications resources as they are unable to reach any processors in the platform model. This scenario is essentially guaranteed to arise in real systems as they will need *some* form of communication with external systems which by definition are not part of the modelled platform.

Communications resources might include:

- On-chip interconnects such as buses or networks.
- Local area network interfaces, such as ethernet or bluetooth.
- Short distance interconnects, such as asynchronous serial (RS232), SPI, PCI express and USB.
- Domain specific interconnects, which in the case of the automotive domain these 'field buses' [180] might include CAN [98], FlexRay [100], SAE J1939 [184], ISO 11783 [99], and LIN [121].
- Interprocessor communications facilities such as shared memory, mailboxes and FIFOs. Regardless of their intended paradigm these resources are still modelled as message passing communications resources.

The broad variety of different possible communications resource types is largely irrelevant to the platform model. The most important property of a communications resource is the set of processors that it can reach. A processor is considered reachable by a communications resource if the resource is able to send a message to the processor *directly*. In this context 'directly' means that the message did not have to be routed between communications interfaces explicitly by any part of the implementation that is dependent on the platform model. For instance, each one of processors E-J in the example platform (figure 3.16) are modelled as being able to reach every other processor in that mesh even though there are no direct channels between them. This is because the routing will be performed transparently by the hardware routers which are not part of the platform model's level of abstraction. A consequence of considering the logical reachability of processors is that the example mesh network has a *complete* PCG, which can be seen in figure 3.15.

The platform model does not define how communications resources should be implemented and the resources do not necessarily need to have a physical representation in

hardware. For example, it is entirely valid to provide 'virtual' communications resources implemented using shared memory or any other hardware storage and communications primitives.

3.3.4 Making Use of the Platform

In the example given in figures 3.14, 3.15 and 3.16 the PCG is small enough to draw a diagram of, and therefore it is realistic that the graph could be held in memory. Large present-day architectures (such as the whole internet) are already too complex to reasonably keep a platform graph in memory even if such a graph could be constructed. Even in the context of comparatively small networks, such as a 10×10 MPNoC, the dense connectivity graph would be of the order of 10KiB[7]. 10KiB may be too large to be useful for the types of resource constrained processors discussed in section 1.5.1, and therefore this represents a conflict between the necessity for information about the PCG, and the space required to represent it.

By considering the cases where the PCG is certainly too large to be useful, such as for an *infinitely* large mesh network, it becomes apparent that a connectivity *graph* is not necessary to reason about the network. A PCG is sufficient but unnecessary to satisfy the queries needed to operate an application on a platform at runtime.

In the context of a specific application (i.e when there is a complete *system*) then there are only two necessary queries that a platform must be able satisfy in order for the application to execute:

Query 1 "which local communications interface is preferable to contact processor *p*?"

Query 2 "which processor(s) could a new machine of type *m* be requested from?"

Clearly, both queries could easily be answered with a complete platform model including the PCG. However, in the case of a regular, homogeneous mesh network then much simpler representations are possible. In the case of homogeneous mesh networks the network interface can contact all other processors and all other processors are equally valid. This simplification is very cheap to represent and would not be demanding to implement in a resource constrained context. However, these query responses are sub-optimal for a variety of reasons including unequal network route and processor load-

[7]Assuming the graph is represented by an adjacency matrix and that no additional memory is required for any representation of the processors

ing, but they are sufficient to enable basic application mapping and do not exclude the possibility of higher quality platform representations where additional complexity is tolerable.

3.3.5 A Platform API

The above queries form the only two requirements on an instance of a platform model: It must be able to determine which communications interface is best to reach another processor, and determine which other processors to request a new machine of a specific type from. This essentially forms an API to query the platform in the implied context of the currently executing processor. The limited expressiveness of these queries provides a lot of freedom in platform representation. There are no requirements to know any specific facts about remote processors, and no requirements to be able to enumerate processors. Some necessary facts about remote processors are implicitly encoded in the queries:

- Query 1 encodes the reachability of a remote processor via a communications resource, as if a communications resource is preferable to communicate with a remote processor, that processor must also be reachable via that resource.
- There is no need to provide a mechanism to enumerate the resources allocated to a processor. If Query 2 responds with a processor that could host a machine of type m, then that processor must satisfy the minimum requirements for that machine. However there is no way to know if the processor still has sufficient spare resources, just that the processor when totally unloaded has enough overall resources to satisfy the request.
- It is not needed to allow processors to be enumerated or 'looked-up' via an address. Processors can only be used for hosting and executing machines, so if a processor is suitable for hosting a machine then Query 2 would already issue it. In the case that a processor cannot host a specific machine then there is not any purpose in having knowledge of its existence as no further action could be made as a consequence of the knowledge.

The platform API only makes sense in the context of a complete system, as the processor-specific subset of the platform definition that can answer these queries must have knowledge of the requirements of the application's machines. Implementations of

the 'framework' component of the MAA (section 3.4) would contain an instance of the platform API.

3.3.6 Realisation of the Platform API

The transformation from the conceptual platform description (with details of all of the processors, resources and interconnect topology) through to processor-specific sets of responses to the platform API is entirely the responsibility of an implementation as there are several viable approaches:

Fully dynamic implementations may chose to retain a complete platform model at runtime on all processors: the platform API is implemented by appealing to the model as needed. The costs of this approach are likely to be prohibitive for any embedded system both in terms of the storage required for the model, and for the computation overheads of querying a complete model on demand. An advantage is that little if any compiler work is necessary to implement this strategy.

Static implementations could generate completely fixed mappings of machines onto processors through the use of static analysis on the application. The platform API would not even need to exist at runtime as the outcome of the API queries would always be the same and would be known with certainty at compile-time, so allowing a compiler complete freedom to optimise away the platform abstractions.

The suitability of an application to fully static compilation depends not only on the construction of the application, but also on the strength of the static analysis technique employed. It is possible in to express applications that conform to the MAA but are intractable to analyse. Some examples of difficult constructions include:

- Requesting a new machine only if some external sensor resource has a particular value at that instant. External input can be assumed to be unknowable, so it cannot be known if that machine will be created, so a fixed mapping of machines to processors is not possible.
- Requesting a new machine only if a message was received with a particular value which can only be determined by executing the code. This condition requires that the application must be executed or simulated in order for its structure to be determined.

Hybrid static-dynamic implementations could use readily available information from the application, such as the machine graph and generate at compile-time a set of constraints on the runtime allocation of the machines. These constraints would assist in making assertions about the runtime properties of the application on a particular processor, such as worst case memory consumption, or worst case periodic event jitter.

3.4 Frameworks and Runtime Behaviour

The framework component of the machine abstract architecture captures the elements of a system that are neither application nor platform specific. Ultimately, the framework component is about the linkage between applications and their eventual runtime platforms. The framework model enables total isolation of applications from any specific platform details by defining a loose coupling between them.

As has already been discussed, application code within a machine can perform three kinds of activities:

- Internal behaviour: performing computation on local state or exclusively owned local resources. This behaviour does not require any wider knowledge of the platform. If application code is executing, it already has the necessary processing, data storage and resource capabilities; the machine does not need to know about anything except what it already has.

- Communication with other machines. This is entirely abstracted by the application model. A machine does not need any platform knowledge to communicate with another machine.

- Requesting new instances of machines. This (potentially) depends on interaction with remote processors, which in turn depends on the platform. This is the primary issue addressed by the framework model.

The framework model is by is by far the smallest of the three system components and specifies only two details:

- Exactly one *start* machine is defined to be created on the platform's *first* processor.

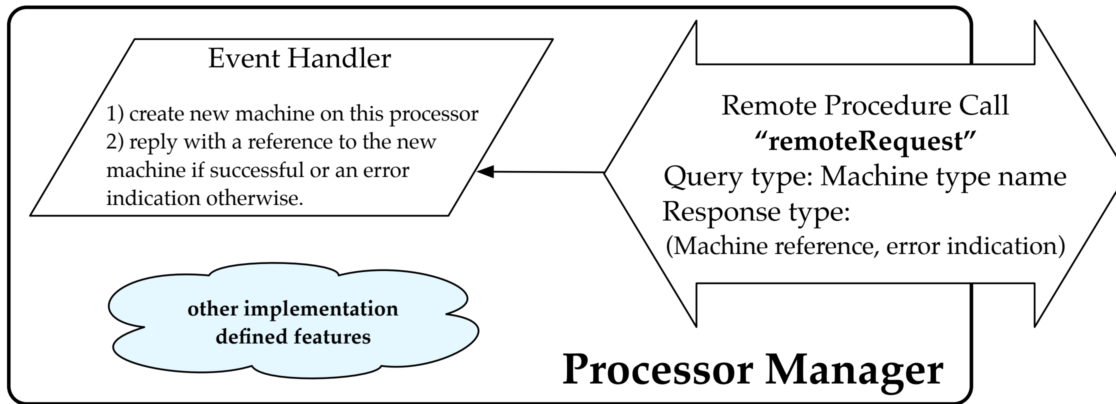


Figure 3.17: *The definition of the **ProcessorManager** machine present on every platform processor. **ProcessorManager**s enable machines to be requested without leaving the machine oriented programming model.*

- Machines are requested indirectly via **ProcessorManager** machines. Every processor in the platform has a **ProcessorManager** instance, and all instances have a identical interface.

3.4.1 Processor Managers

ProcessorManager machines decouple application code from the concept of processors. Their primary and only defined purpose is to facilitate the creation of new machines. Framework implementations must guarantee that every processor in the system has a functional **ProcessorManager** before any other machine issues a request to it; application code can assume that if it has a reference to a **ProcessorManager** that it will actually exist and be able to service requests. The definition of the **ProcessorManager** machine can be seen in figure 3.17.

To request a new machine of type T the following steps must be followed:

1. The executing code should use the platform's API to determine the processors that are *statically* capable of accepting T type machines. The platform implementation will not suggest processors that could never host the specified machine, but the platform does not know the runtime state of a system and can suggest processors that are already 'full'.

2. Decide which processor (p) to use. The executing code has no metric it can use to compare remote processors other than the order the platform API supplied them in. If an implementation has extended **ProcessorManager**s it may be possible to

query a remote processor about its current loading or to determine the network latency or throughput to the remote processor.

3. Send a query to the "remoteRequest" channel in the **ProcessorManager** instance for p. The framework implementation must provide a mechanism for application code to obtain a reference to a **ProcessorManager** for a specified processor. However, framework implementations should not allow processors to be enumerated. A **ProcessorManager**'s "remoteRequest" channel is a **RemoteProcedureCall** where the query type is a machine type name, and the response is a tuple of a new machine reference and an error indication.

4. If the remote **ProcessorManager** was successful it will respond with a reference to the new machine. On failure the implementation defined error indication should be considered to determine the appropriate course of action. In most cases another statically eligible processor can be chosen and the procedure can begin again from step 3.

ProcessorManagers allow machines to be requested using using a machine oriented communications mechanism, therefore this avoids applications communicating directly with remote processors using implementation-specific interfaces. The **ProcessorManager** is necessarily implementation-specific as it must be able to construct new valid machines on the current processor. Without **ProcessorManager**s, a machine would have to communicate directly with a remote processor using a non-modelled communications mechanism.

3.5 Realisation of the Application Model

In order to create a realisation of the machine abstract architecture that is useful for the development of software applications a large number of topics must be addressed:

1. A syntax to capture the static application definition must be defined. Related approaches including Ptolemy II [3, 118] or AUTOSAR system descriptions [18, 29] capture application structure using XML [35] syntax. More dynamic actor oriented approaches, such as Scala [77] and Erlang [104] do not use an external language to define application structure at all; the application is entirely encoded by the application's source code. The *capsule-oriented* 'Panini' [178] programming

language represents a middle-ground where the application is statically elaborated but this structure is defined within the application's source code.

2. A syntax to capture the salient details of the target platform. A complete platform model representation is sufficient but unnecessary for the execution of machine-oriented applications.

3. A programming language must be identified to enable at least the description of the application's event handlers. This alone is a substantial topic and is covered in more detail in the next section.

4. An implementation methodology must be defined to enable the application to become executable on a specific target platform. This methodology will have to address the following issues:

    - A mechanism to provide the isolation and concurrency characteristics of the model. If the selected application programming language contains features that enable data to be shared between execution contexts (such as threads), including as heap objects, global variables, or class variables (static fields), then these mechanisms must be appropriately restricted. The runtime isolation of machines is critical to the arguments for platform independence, scalability and fault tolerance.
    - A strategy for verifying (where possible) that machines interactions are valid. For example, if the programming framework uses different mechanisms for sending to each of the different channel protocols, then the compiler should verify that a sender machine uses the correct mechanism for the target channel's type.
    - A strategy for the enforcement of the defining rules of machine-orientation, including restricting intramachine concurrency to ensure sequential consistency.

3.5.1 Requirements for Programming Languages

A programming language must be able to express a number of the application-model's entities in order to be a viable candidate for machine-oriented programming. The fol-

lowing requirements are for the case where the complete application definition is specified in the code:

- A representation for machine types, and a way to represent their names. In object oriented programming languages classes with particular patterns of organisation would be a natural choice. In less structured programming languages, such as C, `structs` could be used as machine type descriptors. Machine type definitions must contain the following information:
  - The machine type's name. This is used to request new machines of this type. The representation of these names could range from class identifiers (if an machine types are represented as distinct classes), through to application-defined string constants. Ideally these machine type names should be statically analysable to ensure that machine type names can be determined by the compilation workflow.
  - A set of communications channels that this machine type defines. However each of the channel instances are represented, a set of pre-defined protocols should be provided by the implementation framework in preference to allowing applications to define their own communications protocols. Each channel instance in a machine type must have a name defined for it.
  - A representation for statically defined intramachine event sources, such as the startup event source or timed event sources.
  - A definition of the machine's internal state. There are no requirements for the representation of this.
- A mechanism for binding event handler code to their source events.
- A representation for machine references, such as a specific 'machine reference' class or even a string containing an implementation-defined URI [27].
- An implementation of the second platform API (see section 3.3.4): "which processor(s) could a new machine of type *m* be requested from?".
- A set of mechanisms (functions, macros or methods) to:
  - Obtain a reference to the **ProcessorManager** machine for a specified processor, therefore enabling new machines to be requested.

Language	Enforceable?	Embeddable?	Immutables?	Capable?	Realistic?
C	Compile time	✓	✗	✓	✗
C++	Compile time	✓	≈✓	✓	✓
C#	Compile+Run	✗	≈✓	✓	✗
Ada	Compile time	✓	✓	✗	✗
Python	Runtime	✗	≈✓	✓	✗
JavaScript	Runtime	✗	✗	✗	✗
Java	Compile+Run	✓	≈✗	✓	✓

Table 3.5: *A comparison of candidate general purpose programming languages that could be used to implement a machine-oriented programming framework.*

- Send a message to a specified channel (using the channel's name) in a specified machine instance (using a reference to the machine instance).
- Update the machine's internal state.
- Construct new intramachine event sources, such as new `delay` or `yield` event sources.

A speculative evaluation of several general purpose programming languages is summarised in table 3.5. The classifications in table 3.5 are highly subjective and depend heavily on the tradeoffs that are acceptable and the personal bias of the observer. As such, table 3.5 is only a crude illustration of the compared languages' qualities. The table compares following qualities of a programming language:

enforcable describes if the language's standard tooling would be able to enforce any amount of the application model's rules either statically or at runtime. To this end, a language is considered to support compile-time enforcement if it has a compilation stage, and runtime enforcement if it supports dynamic reflection.

embeddable describes the ability of the language to highly target resource constrained embedded architectures.

immutables describes the ability of the language to express and enforce immutable data types. Most of the languages have a **`const`**, **`final`** or similar modifier, enabling an immutable programmatic interface but C and C++ always enable data modification by using a non-const pointer. Likewise, reflection in Java and C# can be used to violate almost any rule.

capable is a crude assessment of the ability of the language to be able to operate an approximation of the machine-model without any modification to the tooling: a library-only implementation.

realistic is a summary of the language, considering its overall suitability for an implementation of a machine-oriented application model in the context of resource constrained MPNoCs. Both C++ and Java have expressive enough languages to represent machine-oriented applications cleanly and have been demonstrated to be suitable for execution on highly constrained processors.

This comparison should only be considered an 'engineering heuristic' as practically any programming language could be used given enough effort or the appropriate use-case. However, Java appears to be a very suitable implementation language but considerable effort is required to achieve an implementation that is compact enough to be useful on limited MPNoC processors.

3.6 Summary

In the previous chapters it was observed that thread-based models of concurrency are a poor match for hardware architectures that have limited or absent shared memory. However, the existing communication-centric programming models were also found to be imperfectly matched to the resource constrained MPNoC domain.

This chapter presented a new actor-oriented programming model intended for the practical description of software for highly distributed embedded systems. Supplementary models for hardware platforms and a model of a runtime framework were also presented to support the application model's goals. Collectively these three models (application, platform and framework) become the *Machine Abstract Architecture*. The minimal framework model enables machine-oriented application code to request new machines without leaving the machine communication abstraction.

Although the application model described is applicable to many distributed problem domains, it is specifically intended to enable implementations to address the challenges of resource constrained MPNoCs. In particular this model is differentiated from existing work by the combination of:

- Explicitly decoupled application and platform models which are also independent of the languages used to describe instances of them. This enables portability of

applications between platforms but also emphasises that an application will eventually execute upon a platform; the application does not define the platform. The application model does not aim to address hardware-software co-design problems.

- Total decentralisation. At runtime the model considers all application machines to be equal. The model does not define any centralised features which would certainly become a bottleneck for a system. Decisions about the allocation of machines to processors are driven by the processor hosting the machine initiating the request for another machine. The model does not require that machines share any state in order to cooperate.

- Explicit support for multiple channels in a single machine each of which may implement a different communications protocol. The benefit of multiple type-safe channels is very clear: channels that consume different data types can be logically related by their owner machine.

- No supervisors, guardians or addresses. The framework does not provide a supervisor entity to manage or monitor machines after their construction, and the model does not impose a supervisor-child relationship between a machine and the machines it requests. These relationships can be implemented in an application if desired but machine addresses (n reference to a machine that does not necessarily exist yet) are fundamentally unsupported by the model and would require substantial work to implement. Additionally, there is no central authority capable of listing or locating machines at runtime. These limitations avoid bottlenecks due to centralised indices, or the complexity implied by a platform-independent distributed index of live machines. Such an extension would be an interesting avenue for future work.

This model retains much of the dynamism of the actor model: machines can request arbitrarily many new machines and exchange references to these machines freely. However, the simple intramachine execution model avoids creating a requirement on a sophisticated operating system or runtime environment.

Chapter 4

Machine Java

Contents

3.1 Machine Abstract Architecture 60
3.1.1 Systems 60
3.2 Application Model 61
3.2.1 Machines 67
3.2.2 Timing, Scheduling and Synchronisation 70
3.2.3 Intramachine Timing and Scheduling 71
3.2.4 Creating Machines 75
3.2.5 Communications 76
3.2.6 Dynamic Synthesis and Communication 95
3.2.7 Resource Usage 107
3.3 Platform Model 107
3.3.1 Processors 110
3.3.2 Resources 112
3.3.3 Communications Resources 114
3.3.4 Making Use of the Platform 116
3.3.5 A Platform API 117
3.3.6 Realisation of the Platform API 118
3.4 Frameworks and Runtime Behaviour 119
3.4.1 Processor Managers 120
3.5 Realisation of the Application Model 121
3.5.1 Requirements for Programming Languages 122
3.6 Summary 125

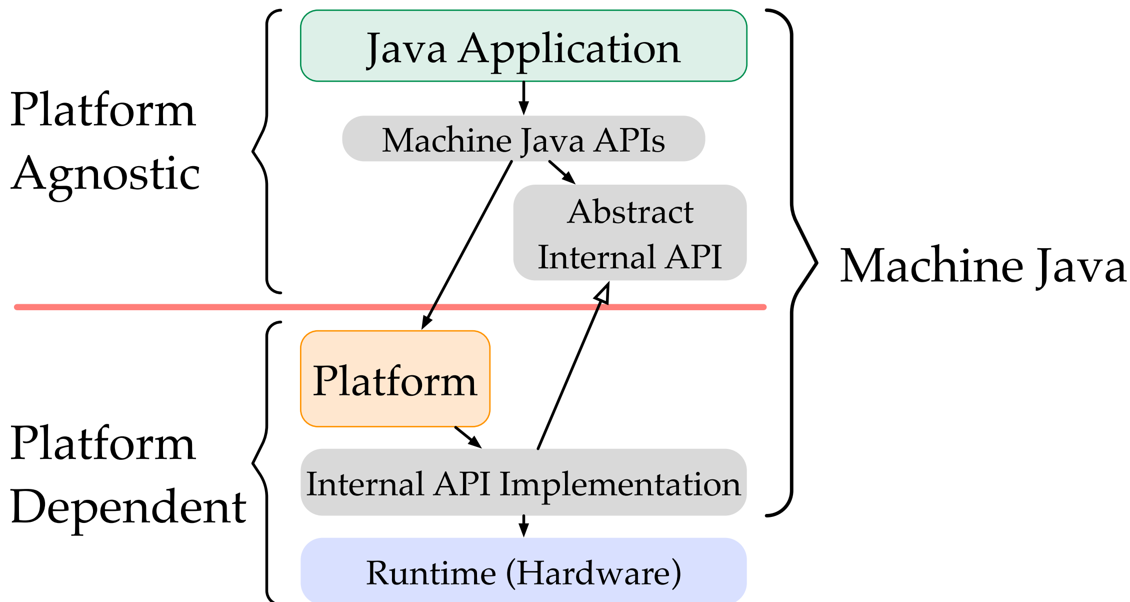


Figure 4.1: *The notional Machine Java stack extends from the user-defined application through to the platform specific API implementations. The lowest tier in the stack is neither defined by the application programmer nor the Machine Java framework.*

In the previous chapter the *Machine Abstract Architecture* was presented including a model for the structure of application software and a model for the representation of distributed hardware. The MAA, although independent of any specific hardware architecture or programming language, is intended to be compatible with the perceived trends in single-chip and tightly-coupled computing.

In this chapter *Machine Java* is presented as a reference implementation for all three Machine Abstract Architecture tiers. In addition to conformance with the MAA, Machine Java provides application-wide static type safety and uses an entirely distributed internal architecture.

The Machine Java stack is illustrated in figure 4.1: Machine Java contains an application programming framework (API) for machine oriented programming, a platform independent implementation of an MAA framework, and an API to allow platform description. Machine Java also defines another layer (the *internal* API) to decouple the implementation of the application API from platform specific concerns. The internal API also decouples the structure of a platform (provided by a platform definition) from the code required to implement Machine Java on that platform. This enables multiple platform descriptions to be created that re-use Machine Java's platform dependent code.

This chapter is divided into three main parts: A discussion of the Machine Java APIs and platform independent framework is provided in section 4.2, the overall theory of

operation is discussed in the context of an example implementation of the internal API in section 4.6, finally the application of Machine Java to bare metal[1] hardware platforms via the *Network Chi* Java compiler is explored in 4.7.

4.1 Java

The *Java* programming language[74] is a natural choice for the implementation of the machine model:

memory safety The Java language has strong memory safety meaning that many common memory access errors (including invalid pointers and buffer overflows) cannot occur[2]. Fundamentally, memory safety is only a desirable trait and not an essential one, but it is the method by which Java implements memory safety that makes it an attractive target language for machine oriented applications: Java uses opaque references rather than integral pointers. Java references cannot be manipulated or cast to any non-reference type. Unlike C code, code written in Java simply cannot assume any underlying structure to Java references and this means a Java runtime has much more freedom over its implementation; Java never has to present the view of a linear memory address space. This abstraction of memory is highly desirable for a machine oriented framework as it ensures that applications cannot be written in a way that is only valid for a specific platform's memory architecture.

type system Java's provides a strong type system with support for parametric types (generics). Strong typing is desirable as it allows static properties of an application to be verified by the compiler, on the condition that these properties can be expressed via the type system. In a machine oriented context it is attractive to exploit a programming language's integrated type system to guarantee intermachine communications compatibility, rather than introduce a 'second-class' type system within the application framework. Additionally, Java already uses a class (`java.lang.Thread`, and subclasses thereof) to represent concurrently executing contexts, so it is not unusual to encode aspects of concurrency within Java's type system.

[1] Platforms without an operating system and related software infrastructure available on every processor.
[2] A good treatment of memory safety topics can be found in [200].

high quality standard library Java has a substantial standard library that applications can assume will exist, including important functionality for mathematics, string handling, data structures (including sets, lists and maps) and common algorithms on these data structures.

ecosystem It has an extensive ecosystem of high quality tools: including development environments[59, 150], runtime and library implementations[68, 152, 151, 89], and compilers for assorted use cases[14, 7, 164]. Due to its significant popularity Java also benefits from active research and technical support communities, open source projects and broad coverage in literature.

Machine Java as discussed in this chapter is an extension to standard Java without any modifications to Java's syntax or virtual machine (JVM). However, while all Machine Java applications are valid Java, the opposite is not true as Machine Java places a number of restrictions on the structure of an application. These restrictions relate to the definition of machine types and the runtime interactions of machine instances, and are discussed in sections 4.3.3 and 4.4.1.3.

4.2 Machine Java

In the following sections Machine Java's realisation of the application model, platform model and framework components are covered:

Machine API to allow machine oriented programs to be developed. This includes the definitions of the *core* classes and interfaces, and discussion of how a machine oriented application can be implemented using the framework, this discussed in the next section.

framework includes implementations of the application model's defined channel types, and the *Processor Manager* concept for requesting remote machine instantiation. This is the platform agnostic component of Machine Java's implementation, and it is discussed in section 4.4.

Platform API allows the essentials of multiprocessor architectures to be expressed in Java code and is discussed in section 4.5.

4.3 Machine Oriented Programming

In brief, Machine Java applications are one or more `Machine` classes with a specified 'starting machine' class to act as an entry point to the application at system startup. `Machine` classes make static[3] declarations of their communications interfaces and are programmed in an event driven style. All non-startup behaviours in a `Machine` class are defined by *event handlers* in the machine, which can be triggered at runtime by any of the machine's *event sources*. Machine Java's event sources are a very general concept: In addition to communications interfaces, timing utilities (such as delays, alarms and periodic execution) are also abstracted as event sources. Only the *intermachine* subset of event sources (that represent interaction between machines) are required to be statically declared, whereas *intramachine* event sources are free to be instantiated dynamically. Table 4.1 provides a summary of the mapping between application model entities and their representation in Machine Java.

This section addresses the application and machine API layers of the Machine Java stack.

One of the most important aspects of Machine Java is that both *live* machines and machine references are represented identically by machine objects (instances of subclasses of `Machine`). Application code cannot differentiate between a machine object that represents a live machine on the current processor and a reference to a machine on a remote processor. Applications do not have to treat machine objects specially; they can be stored in data structures and 'lost' (unreferenced in the application) just as any other object can be in Java. As implied by the application model, machine objects can even be sent via communications channels.

[3]In this context static means bound at compile-time, rather than Java's **`static`** storage class.

Entity	Machine Java Representation	Example
machine types	Subclasses of Machine	**class** FlowControllerextends Machine {...}
machine type names	Java Class<M **extends** Machine> objects	Class<FlowController>
machines	Machine objects	**private** FlowController fc;
machine references	Machine objects	**private** FlowController fc;
channels	Implementations of Intermachine interface	**new** BoundedBuffer<Integer>(...)
channels names	Identifiers for **public** machine fields	**public final** Signal<Integer> setLevel;
event handler	Handler<T> objects	Handler<Integer> h= **new** Handler<Integer>(){...}
mutable data	Any ordinary Java data type	**private int**[]history;
immutable data	Immutable data types	**final** ImmutableSet<Integer> latest;

Table 4.1: *The Machine Java representations of the important application model entities.*

Machine Java applications combine the structural definition of an application with the code that will provide runtime behaviour and therefore Machine Java avoids a dependency on a separate description language for an application's static characteristics. An application's structure of `Machine` classes and communications channels that it defines can be interpreted as an application-specific virtual platform where each machine is a virtual processor and the channels between them are the virtual interconnect.

Hello World!

As a first example, a 'Hello World' application can be seen in figure 4.2. When executed in a standard Java Runtime Environment (JRE) Java's usual entry point[4] (**`static`** `main (...)`) is the first application code to be executed. The `HelloWorld` class extends Machine Java's `Start` class, which in turn extends the `Machine` class. Subclassing `Start` marks `HelloWorld` as both a machine type and a possible entry point for a machine oriented application. This allows `HelloWorld`'s `main` method to bootstrap the Machine Java framework by invoking the **`protected static`** `start()` method defined within the `Start` class. From this point the Machine Java framework coordinates code execution; the `start()` method is not required to ever return control. Once the framework has initialised, a single instance of the `HelloWorld` machine is created on the current platform's *first* processor where it will begin to execute its `internal()` method. `HelloWorld.internal()` calls the `log()` helper method provided by the `Machine` class, which writes a message to whatever logging mechanism is available.

After execution the following text is available in the log:

```
HelloWorld@0,0:1> Hello world!
```

The output text betrays a little runtime detail: The "`@0,0:1`" indicates that this `HelloWorld` machine was executing on the processor at coordinates $(0, 0)$ and was the second machine to be created on that processor. The first machine on any processor is always a **ProcessorManager** (section 4.4.4). This logged detail is specific to the implementation of Machine Java's internal APIs, and the current platform implementation. In this case the 'XYNetwork' platform (see 4.5) for mesh networks is in use, and this then depends on the 'networkchi' implementation (see 4.6) of Machine Java's internal APIs. The

[4]Any Java class can contain an entry point method, so the JVM is instructed which class contains the entry point of interest. This is usually handled transparently by development environments or application loaders.

```java
package examples.helloworld;
import mjava.core.Start;

public class HelloWorld extends Start {

  public static void main(String[] args) {
    //start(...) is a static method defined in Start.java
    start(HelloWorld.class);
  }

  private HelloWorld() {}

  @Override
  protected void internal() {
    log("Hello world!");
  }
}
```

Figure 4.2: *A complete 'Hello World' application in Machine Java. The Java entry point (`main()`) executes first and starts the application. This causes a `HelloWorld` machine to be created which will then log a "Hello world!" message.*

'networkchi' implementation enables operation on embedded processors and networks supported by the Network Chi Java compiler (see section 4.7). A detailed explanation of Machine Java's startup sequence, from `main()` through to the execution of the start machine's `internal()` method is provided in section 4.6.1.

The 'hello world' application highlights the basic style of Machine Java code, but only covers the most basic aspects of application startup and debug logging. The other core concepts, including event handlers and intermachine channels are discussed in the subsequent sections.

4.3.1 Single-Thread Equivalence

The application model makes strong requirements on the timing and scheduling of intramachine code execution, most notably that it is sequentially consistent (see 3.2.3). Sequential consistency is provided in Machine Java through *single-thread equivalence*. At runtime all `Machine` objects execute concurrently but within a machine the code is executed with the same semantics as standard single-theaded Java. `Machines` will execute as if only one thread exists in the Java runtime; no code within a machine can ever be observed to execute concurrently. Standard Java threads (provided by `java.`

`lang.Thread`) are not available to Machine Java applications as their use undermines the guarantees that the application model is intended to provide.

Most complications of multi-threaded programming relate to maintaining the consistency of data structures across concurrent accesses, and these consistency issues are avoided by Machine Java as concurrent access to data structures cannot be expressed; machines are guaranteed to only have one internal execution context and data structures cannot be shared between machine instances. Single-thread equivalence within a machine also has the following implications:

- The **`synchronized`** keyword for entering an object's monitor is unnecessary and has no meaning in Machine Java. There is no situation in which two or more machines could hold references to the same object so Java's monitors provide no utility.
- Java libraries to assist with concurrency (such as the `java.util.concurrent` package) should be avoided. These libraries are designed for the standard Java thread concurrency model and provide little if any utility to a machine-model application.
- The thread safety of third party Java libraries is not an issue as they can never be executed in a multi-threaded context, but this does also imply libraries that dependent on threading for performance or correctness must be re-architected for the machine model. This is unlikely to be a concern in embedded systems as the constrained resources of these platforms largely prevents the re-use of existing general purpose libraries.

4.3.2 Machine Classes

In Machine Java, subclasses of the abstract `Machine` class represent the application model's machine types. While a machine is executing code its activity always falls within one of the following categories:

internal behaviour Operating on data contained within the machine instance, or interacting with a resource used exclusively by the local instance is *internal behaviour*. APIs to interact with resources are not provided by Machine Java; only the ability to express that a machine will require a resource is provided. Application code can use whatever mechanism their JRE or compiler supports to use a resource, such as:

```
public class BasicSender extends Start {
  private BasicSender(){}

  public static void main(String[] args) {
    start(BasicSender.class);
  }

  @Override
  protected void internal() {
    BasicReceiver rx = newMachine(BasicReceiver.class);
    rx.numbers.send(0x1337);
  }
}
```

```
public class BasicReceiver extends Machine {
  private BasicReceiver(){}

  public final Slot<Integer> numbers = new Slot<Integer>(new Handler<
      Envelope<Integer>>() {

    public void handle(Envelope<Integer> info) {
      log(info.getPayload());
    }
  });
}
```

Figure 4.3: *A basic application with two machines. The start machine, `BasicSender`, creates an instance of `BasicReceiver` and sends the value `0x1337` to the new machine.*

Java Native Interface[17] (JNI), sockets, or raw memory access[5]. Internal behaviour is not the concern of Machine Java, and therefore no internal behaviour *requires* the use of any Machine Java APIs. However, Machine Java does provide APIs to assist with intramachine timing, which is possible but awkward and inefficient to achieve otherwise. These are discussed further in section 4.4.3.

intermachine interaction Machines can interact with any of the other live machines, on the condition that the machine initiating the interaction owns a reference to the

[5]Standard JREs support breaking memory abstraction via `sun.misc.Unsafe`, and Chi (see section 4.7 provides a similar library (`chi.runtime.Unsafe`) for direct access to memory.

remote machine. All machine interactions happen through `Intermachine` channels which are defined defined by subclasses of `Machine`. A machine defines its channels by creating **`public final`** fields that are initialised with instances of `Intermachine` classes. It is possible for an application to use Machine Java's internal APIs to create new channel protocols; this is not required as Machine Java provides the suite of standard protocols defined by the application model (see 3.2.5.3 and 4.4.1). Figure 4.3 shows an example of a basic communication between two machines. The start machine `BasicSender` requests a new machine and then uses the reference it has been provided to send a message to the new machine. In `BasicReceiver` a single '`Slot`' channel is defined. `Slot` channels are non-overwriting (meaning that senders can only send when buffer space is available), and have a buffer length of one data item.

In `BasicReceiver` the `numbers` channel is provided with an event handler that will be invoked whenever the `numbers` channel receives a new data item, in this case an integer. In Machine Java event handlers are implementations of the `Handler<T>` generic interface, which has one method: `void handle(T info);` where T is a type that is useful to the event in question. For the `numbers` channel in `BasicReceiver`, the handler's type is `Envelope<Integer>` as all unidirectional channels wrap received data items into `Envelope` objects. On execution the application in figure 4.3 logs the following:

```
BasicReceiver@1,0:1> 4919
```

This indicates that the `BasicReceiver` was *not* created on the first processor at $(0,0)$, but was allocated to the processor adjacent at $(1,0)$. `BasicReceiver` has no spontaneous behaviour, it only executes code in response to an integer arriving to its `numbers` channel.

API interaction The final category of activity within a machine is interaction with the Machine Java framework via an API. All interactions with the framework occur through methods declared within the `Machine` class itself; there is a single 'point of contact' between application defined code and the Machine Java framework. Machine Java exposes very little functionality to machines: just enough to request new machines and to log messages.

```java
public abstract class Machine {
  public static Machine getThisMachine() {...}

  //May be overriden by subclasses
  protected void internal() {}

  //Three methods to request new machines:
  public final <T extends Machine> T newMachine(Class<T> type) {...}

  public final <T extends Machine> void newMachine(Class<T> type,
     Handler<T> machineArrivedHandler) {...}

  public final <SetupValueType,T extends SetupableMachine<SetupValueType
     >> void setupMachine(Class<T> type, final SetupValueType
     setupValue, final Handler<T> machineSetupHandler) {...}

  protected void setPriority(int priority) {...}

  public String toString() {...}
  public int hashCode() {...}
  public boolean equals(Object obj) {...}

  public void log(Object message) {}

  //Only callable within a machine's constructor:
  protected void REQUIRE_RESOURCE(Resource res) {...}
  protected void ENSURE_SINGLETON() {...}

  //For framework use:
  public MachineDriver<? extends Machine> getMachineDriver() {...}
}
```

Figure 4.4: *An abridged version of the `Machine` class to outline the functionality provided to machine implementations.*

The API available to subclasses of `Machine` is provided in figure 4.4. This API has a few notable features not already covered:

- `Machine.getThisMachine()` (line 2) is **`public`** and **`static`**, so it can be called from any code in an application and not just from within subclasses of `Machine`. This method returns a reference to the machine that is the context for the code that is currently executing. The enables *machine aware* libraries to be constructed which can request their own worker machines, or perform logging via the current machine context.

- `internal()` is invoked in a machine once it has been fully initialised but before all other event handlers. This is the *startup* event described in the application model (section 3.2.3.1).

- In Machine Java applications do not communicate with remote **ProcessorManager** (see 4.4.4) machines directly. Instead `Machine` provides three distinct helper methods to request a new machine (lines 8, 10 and 12), each with different semantics: a blocking request, a non-blocking request and a request with initialisation data. These are discussed more thoroughly in section 4.3.4. These methods further isolate application machines from the concept of a 'platform' and even the existence of non-local processors.

- Machines themselves have a priority which can be used by implementations to determine the allocation of implicit resources, such as CPU time and memory. This priority is a positive integer where lower numbers are higher priorities.

- Three of the methods defined by `java.lang.Object` are overridden with implementations that make more sense for machines. `toString()` returns the name of the machine class with a brief platform-dependent description of the machine's location, `hashCode()` and `equals()` follow the contract a programmer would expect: two machines are equal according to `equals()` iff the two machine objects refer to the same machine instance.

- Two methods can only be called within a machine's instance constructor: `REQUIRE_RESOURCE(...)` and `ENSURE_SINGLETON()`. These two methods throw a runtime exception (`ResourceUnavailableException`) if the construction of the machine would violate resource constraints on the processor that is attempting

to create the machine. `ENSURE_SINGLETON()` asserts that the machine being constructed is a singleton (no other machines of the same type exist) on the current processor. These methods are capitalised to represent that they are essentially static preconditions for a machine, but happen to be executable code too. It is easy to determine analytically if either of these methods are called from a machine's constructor by considering an application's method call graph. This allows resource requirement and singleton preconditions to be addressed where possible at compile time but also enforced at runtime.

The machine class also contains methods to provide access to Machine Java's internal APIs and to allow *event sources* within a machine to schedule events.

4.3.3 Machine Class Restrictions

Following from the application model, in Machine Java machines are the *only* context for application code execution and data storage. This has some important implications:

code execution if there is code executing in an application it must be executing in the context of exactly one machine. This means that all control flows in an application must have started from a specific machine's event scheduler and are confined to the context of that machine. Other execution contexts may actually exist at runtime but these can never be visible to application code. The practical consequences for `Machine` subclasses include:

- There can be no intermachine method calls; A machine cannot invoke a method in any other machine instance, even if it is the same type. This is partially enforced by requiring that all methods defined by a `Machine` subclass are marked with either of the **`private`** or **`protected`** access modifiers. Java allows objects to invoke private methods in other object of the same type, so this restriction only provides partial protection. Machine Java does guarantee that machine isolation will never be violated by the invocation of an instance method in another machine. This is discussed further in section 4.6.

  Static methods in machine classes can always be public as they do not have access to instance fields, and static fields are highly restricted in machine types (see below). It is possible to *apparently* violate machine isolation by defining a static method in a machine that takes an instance of the enclosing

```java
public class DataMachine extends Machine {
  private DataMachine() {};

  private int secretData;

  //Won't work unless 'instance' is the calling machine
  public static int getSecretData(DataMachine instance) {
    return instance.secretData;
  }
}
```

Figure 4.5: *The definition of `getSecret` is allowed according to the rules for valid machine types but appears to violate machine isolation. Machine Java guarantees that machine isolation cannot be violated by 'accessing' the instance data of other machines.*

type as a parameter and then returns a private instance field from the passed machine object. This construction is shown in figure 4.5. In all cases Machine Java guarantees that cross-instance method calls will not violate machine isolation, but this very likely also guarantees that such invocations will not work as expected and should be avoided where not explicitly restricted.

- Machines cannot be interrupted, there are no asynchronous transfers of control (ATC) and the runtime will never invoke code within a machine from any non-machine execution context.

data storage If an object is reachable by application code within a specific machine then the graph of references that reaches the object can only be rooted in a local variable currently in scope of currently executing code, or it must be rooted in an instance field of the same live machine. More concisely: at most one machine can ever reach the same object. Additionally it follows that if no machine can reach an object then it can never be accessed ever again and is eligible for garbage collection.

This is slightly different to standard Java which allows objects to be accessed from all execution contexts (threads), and therefore objects are considered reachable if they are referenced by an in-scope local variable, a live thread, or a static field of a class. The practical consequences for `Machine` subclasses include:

- Machines cannot define **`static`** fields. This would allow data to have a lifetime beyond any instance of that machine type, and importantly it would allow multiple machine instances to reach the same object and this would

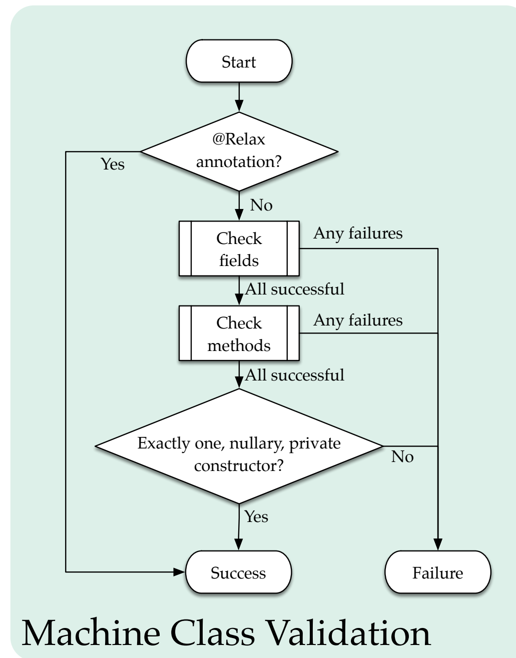
violate the isolation of machines. This policy is enforced by the Machine Java framework for Machine classes, but not for library classes. In this case it cannot be expected that `static` fields will behave as they do in standard Java. Implementations of Machine Java's internal APIs can define the precise behaviour of `static`s, ranging from conversion to implicit instance fields for each Machine that can call code that can access the field, through to outright prohibition on `static` fields in library code. The 'networkchi' internal API implementation (section 4.6) maintains separate storage of `static` fields for each processor, potentially allowing library code to violate the isolation of machines within a processor. However, as application code cannot directly influence which processor a machine might be allocated to at runtime there is little opportunity for an application to deliberately exploit this relaxation of machine isolation.

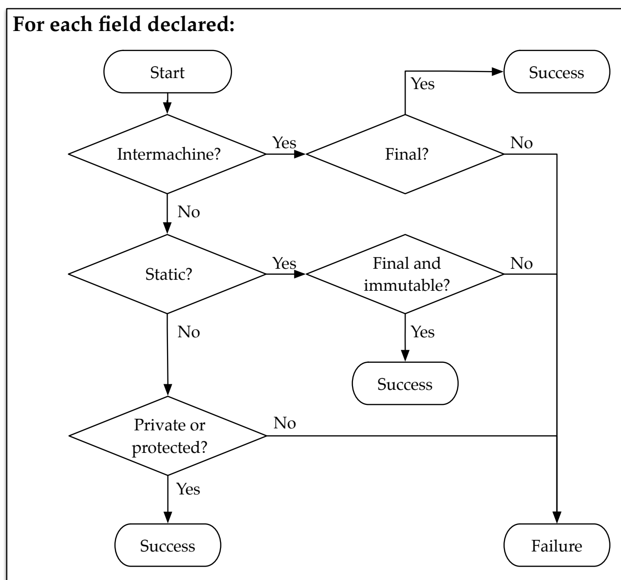
As an exception to this rule, `static final` fields are permissible if they are of an immutable type (see 4.4.1.3) as this retains intermachine isolation.

- Machines must protect all non-static non-intermachine[6] declared fields from external access by marking them with the either of the `private` or `protected` access modifiers. Even final immutable fields must be access protected. This restriction does not itself guarantee intermachine isolation, as Java objects can access private fields of other objects if they are the same type. Likewise, the `protected` access modifier grants access to subclasses. However, machine Java *does* guarantee that machine isolation cannot be violated by accessing private or protected fields of another machine instance. This mechanism is described in section 4.6.

The validation procedure for machine classes is shown in figure 4.6. Machine subclasses annotated with `@Relax` always pass validation and can therefore contain restricted patterns of fields, methods and constructors. `@Relax`ed machines are just as strongly isolated at runtime as all other machines, so the `@Relax` annotation is just an expression that the the programmer understands the machine execution model and does not require as much tool assistance. `@Relax` is primarily useful for rapid prototyping and sophisticated application architectures involving Java interfaces (which can only

[6]Channels are intermachine classes so are permitted to be declared `public final`.





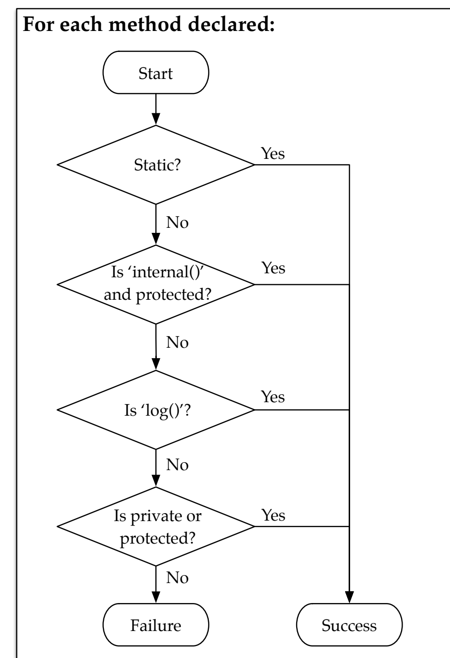


Figure 4.6: *The procedure for determining if a machine class conforms to Machine Java's rules.*

define public instance methods).

4.3.4 Machine Life Cycle

At system startup the Machine Java framework creates a **ProcessorManager** machine on every processor defined by the current platform. If the processor is also the *first* processor, the application's *start* machine will also be created and have its `internal()` method queued for execution in the machine's event queue. All other processors will wait, idle, until their **ProcessorManager** receives a request for a new machine.

Application is defined in section 3.2 as the pair of a set of machine types and a specific machine type to *start* the application. The application's start machine must extend the `Start` abstract class to denote that it is a machine capable of starting an application, but this is not sufficient on its own to define an application though, as an application may contain multiple machines *capable* of starting an application but not used in that capacity. The invocation of the **`public static`** `start()` method (defined in the `Start` class) requires the class of a particular start machine as a parameter, and this is what unambiguously identifies the start machine for a Machine Java application.

Start machines must contain some spontaneous behaviour, such as an implementation of the `internal()` method or a timed event source that will trigger at some point in the future. Without some spontaneous behaviour the application will never make progress as (by definition) there are no other machines in existence that could trigger behaviour in the start machine. From this stage in an application's life cycle, all machines are created at the request of application code.

Machine objects cannot be instantiated in application code using Java's **`new`** operator but instead they have to be requested from the framework. Java's **`new`** operator only allocates memory and executes the specified constructor in the current execution context, so a machine object created using **`new`** would both be inert (no new execution context) and would certainly be allocated to the current processor. As the semantics of the **`new`** operator are not appropriate for machine instances, all machines must protect their constructors from use by making them private. This rule can be seen in the machine type validation procedure (figure 4.6).

Machines are required to declare all intermachine fields (channels) as **`final`** to prevent the machine from attempting to replace a channel implementation at runtime. This means that all channel fields must be initialised before a machine's constructor has fin-

ished executing. The clearest way to initialise channels is by using Java's *field initialisers*, such as the one shown on line 4 of figure 4.7. Machines must also declare all channel fields **`public`** to allow other machine classes to access the fields that *name* the channels.

A machine's constructor cannot interact with other machines as:

- It does not start with any machine references.
- Other machines cannot get a reference to it until construction has finished.
- It cannot request a new machine and gain a reference before the constructor has finished. (See below)

New machines can be requested via one of three methods defined (see figure 4.4) in the `Machine` class:

```
T newMachine(Class<T> type){...}
```

Is a *blocking* request for a machine. This method returns a reference to the new machine class directly, but it does not return until the machine has been created and is ready for use. This is by far the easiest API for requesting new machines as the program flow is simple and the method will throw an exception if the new machine could not be created for any reason. However, blocking the sender is not ideal as this prevents all activity in the machine until the method returns. The method guarantees static type safety for the caller as the method only accepts classes that represent machines, and is defined to return a machine of the same type as the class supplied represents. This method must not be used during a machine's constructor as this introduces the possibility of application deadlocks. This is because machine constructors are invoked within an event handler in the processor's **ProcessorManager** machine, meaning that other requests to the same **ProcessorManager** cannot be serviced until the new machine's constructor has finished. The following sequence of events would deadlock an application:

1. Machine *a* on processor 0 requests a new machine, *b*, in an ordinary event handler. The request is dispatched to processor 1.
2. The **ProcessorManager** on 1 handles the event caused by the request from 0. During its processing it generates a new execution context and invokes *b*'s constructor.
3. *b*'s constructor uses the blocking `newMachine` to request machine *c*, this request is dispatched to 2.

4. The **ProcessorManager** on 2 handles the event caused by the request from 1. During its processing it generates a new execution context and invokes c's constructor.

5. c's constructor uses the blocking `newMachine` to request machine d, this request is dispatched to 1. This request never finishes as the **ProcessorManager** on 1 is stalled waiting for b to finish construction. There is now a circular chain of dependencies and (this section) of the application is now deadlocked.

```
void newMachine(Class<T> type, Handler<T> machineArrivedHandler){...}
```

Is a non-blocking request for a machine. This method returns immediately but can accept an optional second parameter for an event handler. When the machine has 'arrived' (it has been constructed, potentially on a remote processor, and is ready to use) the specified handler will be put into the machine's event queue. A reference to the new machine is supplied as the parameter to the event handler when it is invoked. This method of requesting a machine is much more efficient as the requesting machine can continue its operation as usual, at the expense of the complexity of another event handler. Event-driven machine requests are a more natural fit for a machine oriented programming style. The primary benefit of this method is that it avoids the machine-request deadlock scenario described previously, but it has the significant disadvantage that it cannot throw an exception if unsuccessful: The event handler is not typed to accept an exception instead of a machine. If there is a problem with the request the arrival handler will be invoked with a **`null`** machine argument. Future refinements could improve this API with a second event handler for unsuccessful machine requests, or supply a typed pair to the event handler (`Pair<T, Throwable>`) where either the machine is non-null for success, or the `Throwable` is non-null if unsuccessful. Machine constructors *can* use this method (and the method below) to request a new machine, but the `machineArrivedHandler` will only be invoked *after* the machine's constructor and `internal()` method have finished.

```
<SetupValueType,T extends SetupableMachine<SetupValueType>>
void setupMachine(Class<T> type, final SetupValueType setupValue, final
   Handler<T> machineSetupHandler) {...}
```

```
public abstract class SetupableMachine<T> extends Machine {
  protected SetupableMachine(){};

  public final Signal<T> _setupBuffer = new Signal<T>(new Handler<
      Envelope<T>>() {
    public void handle(Envelope<T> info) {
      setup(info.getPayload());
      //only ever receive once:
      _setupBuffer.shutdown();
    }
  });

  //must be implemented by subclasses:
  protected abstract void setup(T value);
}
```

Figure 4.7: *`SetupableMachine` is a variant of `Machine` with the ability to receive initialisation data of a specific type. Note that the `_setupBuffer` is closed in its event handler, ensuring that at most one setup item will ever be received.*

This somewhat complex[7] method is the third API to request a machine. It is a non-blocking request and can supply some initialisation data to the new machine. The application model describes (in section 3.2.4) that new machines cannot be created with any parameters; all new machines of the same type are identical, 'empty' and their existence cannot be predicated on some supplied data. However, after a machine has been created it can receive data via a channel as usual. This method creates a machine as if by using the non-blocking `newMachine` API, but sends the supplied `setupValue` to the new machine before a reference to the machine is returned to the application. This method can only request machines which extend the `SetupableMachine` (see figure 4.7) class of machine. `SetupableMachine` defines a `Signal` channel which is typed to accept the setup data supplied. Subclasses of `SetupableMachine` must implement the `setup(T ...)` method to accept the initialisation data item.

4.3.5 Machine End-of-Life

The application model explains (section 3.2.4) that a machine, m, cannot be destroyed at the request of another machine, r, as r cannot be sure that m is universally unneeded. This follows from the concept of machine logical locality: a machine cannot make correct

[7] The full type signature has been retained here for 'clarity'.

statements about the application outside of its logical local area.

Machine Java does, in some limited circumstances, support the destruction of machines. Machine Java will automatically destroy a machine if the following are all true:

1. All channels in the machine are *closed*.
2. **and** the machine is not waiting for a reply to a bidirectional request.
3. **and** the machine has an empty event queue.
4. **and** the machine has no timed event sources that can still trigger an event.

As soon as all of these conditions have been satisfied the machine is already considered dead; it can never execute any more event handlers and it is therefore eligible for destruction at a time convenient to the framework. Condition (1) is stronger than it appears as not all channel types are closable which implies that a machine can only ever be destroyed if all of its channels are of *closable* types. Machines which only define closable channels are considered *ephemeral*, and only unidirectional destructive channels (such as the `Signal`, shown in figure 4.7) are closable.

4.3.5.1 Simple Ephemeral Machines

The most basic ephemeral machine has no channels at all, and as it can never communicate with existing machines, it could only be useful as an application's start machine. The simplest useful ephemeral machine can extend `SetupMachine` with no additional channels or spontaneous event sources defined. It can receive any machine references necessary in its setup data. This construction is safe as a *Simple Ephemeral Machine* (SEM) can only ever initiate interaction with other machines: It is not possible for other machines to have a useful reference to a SEM as it has no usable channels. Condition (2) guarantees that a SEM will not be destroyed during a bidirectional request that it initiates, so a machine responding to a query from a SEM can be confident that the response is being directed to a live machine.

Simple ephemeral machines can be used as efficiently as any other machine in work-pooling architectures, but work must be fetched from a pool machine rather than pushed to it. An idle worker SEM does not need to periodically poll its pool manager for work, which would waste network and processor capacity, instead it can request work from an ordinary `RemoteDataFetch` channel defined by the pool manager. The key is

that the pool manager delays its response to the work request until it has something useful to communicate to the worker SEM, which could include work items, a machine reference for a new pool manager, or a notification to shutdown. Ordinarily the worker SEM would perform the action and fetch another item from the pool manager, but if requested to shut down it simply does no further requests. Once its `RemoteDataFetch` response event handler is concluded the machine will be eligible for destruction.

Applications must be designed carefully if it is possible for a non-SEM to become dead while it is still 'in use' by some other machines. This situation is not a severe problem as data sent to a dead or destroyed machine is just discarded, the application will continue to run as normal but the initiating machines will obviously never receive a response. In addition, condition (1) ensures a machine that explicitly defines a channel will never be destroyed automatically until the channel is deliberately closed by application code; programmer error is required to destroy a machine that is still needed.

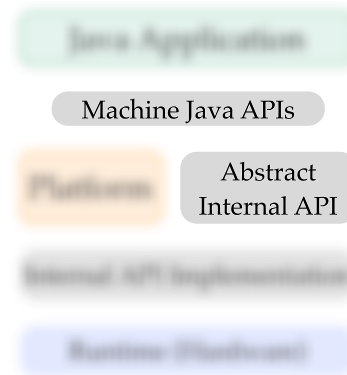
It would be possible for more sophisticated implementations of Machine Java to use a form of distributed garbage collection in order to determine which machines are certainly unused, and this would enable automatic destruction of machines with open channels too. However, the 'networkchi' (section 4.6) implementation of Machine Java's internal APIs is only capable of machine destruction in accordance with the conditions above.

4.4 Platform Agnostic Framework

In addition to the `Machine` class and its two more specialised descendants, `Start` and `SetupableMachine`, Machine Java provides (and demonstrates the implementation of) a platform independent library of event sources. Both intermachine communications channels (section 4.4.1) and spontaneous event sources (section 4.4.3) can be constructed effectively in Machine Java. A realisation of the **ProcessorManager** concept is also provided (section 4.4.4) to allow all application-defined interprocessor communication to be confined to the machine abstraction.

4.4.1 Communications Channels

Machine Java provides implementations for many of the useful protocols defined in the application model (section 3.2.5.3). An overview of Machine Java's predefined protocols and their relationships can be seen in figure 4.8. As the protocols were already discussed at length in section 3.2.5.2, this section discusses the general representation of channel protocols in Machine Java applications. In the subsequent section (4.4.2) the implementation of channel protocols using Machine Java's internal communication API is explored.



This section discusses the APIs available to Machine Java applications and their platform-agnostic implementation using internal drivers.

In Machine Java the suite of channel implementations and supporting libraries forms the *Two Party Interaction Framework* (TPIF). An overview of this framework's components and important relationships is provided in figure 4.8.

In Machine Java, machines interact with each other through 'channels' assigned to **`public final`** fields in machine classes. A field can only be made **`public`** in a machine class if its type implements the `Intermachine` interface. The `Intermachine` interface is purely symbolic; it exists to allow a class to indicate that it will not violate machine spatial isolation and is safe to be made public. Channels provide two major kinds of functionality within a machine: they are *event sources*, and they can coordinate data exchange between machines.

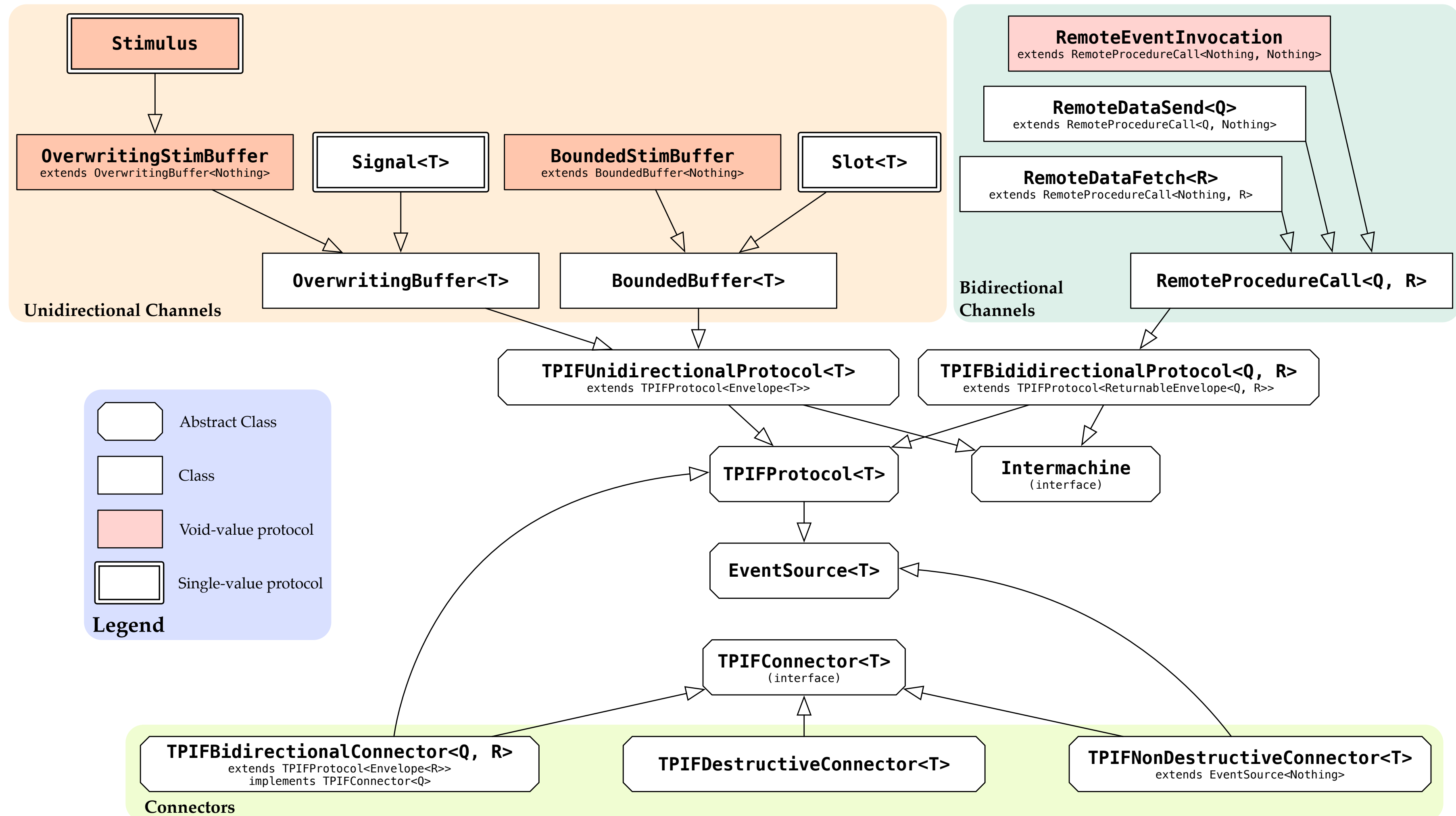


Figure 4.8: *An overview of the **Two Party Interaction Framework** in Machine Java. This library provides machine oriented applications with many of the channel protocols discussed in section 3.2.5.3. Every protocol also has a subclass of a connector associated with it but these have been omitted for simplicity.*

At this point it is helpful to cover some deeper details of Machine Java's implementation: To be an event source, a class must extend the abstract `EventSource` class shown in figure 4.8. A simplified version of `EventSource`'s Java definition can be seen in figure 4.9. An event source is granted the ability to register events into the machine's event queue, and importantly for channels, event sources are provided an *Intermachine Event Source Identifier* (`IESIdentifier`). A `IESIdentifier` uniquely identifies the specific combination of a channel and the machine instance it is contained in. All event sources are generic on the type of object that will be passed to the application's event handlers for that source. For event sources that provide no data, Machine Java defines the `Nothing` class. `Nothing` is essentially a *void* type but it can be used in situations where **`null`** is forbidden. Java's `Void`[8] class is insufficient in these situations as **`null`** is the *only* value which can be assigned to a `Void` typed variable. Machine Java's `Nothing` is also uninstantiable but a single universal instance (`Nothing.NOTHING`) is available as a placeholder.

So far these event source details are only to make coding channels slightly easier. However, the most important (and eventually interesting) feature of the `EventSource` class is that it captures the machine instance that *owns* the event source in its `owner` field.

As far as possible Machine Java conforms to the application model, and therefore it provides strong guarantees of consistency, isolation and platform independence to application code, but this is not the case for implementations of framework components including event sources. Machine Java only provides isolation and platform independence to event source implementations, meaning that the execution context may be (and often is) *inconsistent*. Code within event sources will sometimes be executed in machine contexts other than the machine that apparently created the object, and in some cases code may execute in no machine context at all. For this reason, event source implementations must be especially careful to avoid assumptions about their current execution context to avoid violating the consistency of the machine which created the event source. In practice this means that event sources must *never* invoke application code directly, as this could result in a violation of single-thread equivalence within the machine; all application code must be executed via the event queue. Event sources must also be careful to refer to their `owner` field to determine the machine that constructed them, as after

[8] `java.lang.Void` is Java's object representation of the **`void`** pseudo-primitive. No value for **`void`** can exist and therefore there are no instances of `Void`.

```
1  public abstract class EventSource<T> {
2    public final Machine owner = Machine.getThisMachine();
3    protected Handler<? super T> _handler;
4
5    public EventSource(Handler<? super T> handler) {...}
6
7    public void setHandler(Handler<? super T> handler) {...}
8
9    //register events:
10   protected void registerEvent(T info) {...}
11   protected void registerEvent(Handler<? super T> handler, T info)
         {...}
12
13   //For Intermachine event sources:
14   protected IESIdentifier getExternalID() {...}
15
16   //manage priority:
17   protected int getPriority() {...}
18   public void setPriority(int priority) {...}
19 }
```

Figure 4.9: *An abridged version of the EventSource abstract class. The EventSource class provides functionality for recording event handlers, setting event source's priority, and registering events into the machine's event queue.*

Attribute	Destructive Channels	Non-destructive Channels
Can refuse sends?	No	Yes
Send-side event source	No	Yes
Send-side event type	–	Uni[9]: `Nothing`
		Bidi[10]: defined response type
Buffer-full policy	Last-is-best overwrite	Refuse receives
Closable	Yes	No

Table 4.2: *A comparison of the differences between destructive and non-destructive channels in Machine Java.*

construction `Machine.getThisMachine()` will often yield a machine object that does not match the owner machine.

Although the application model distinguishes between non-destructive and destructive channel read operations, Machine Java does not: Channels in Machine Java do not have methods to read data. The only way for a machine to get data from a channel is by handling the arrival event for the data item. The distinction between destructive and non-destructive *writes* is important though, and their characteristics can be seen in table 4.2.

It can be seen in table 4.2 that non-destructive channels are also event sources in the machine that sends data. To accommodate this Machine Java defines the concept of channel *connectors*. The three types of connector can be seen in the TPIF overview figure 4.8. To communicate with a remote channel, a machine must refer to the channel and request a new connector. It is this connector object that is the event source in the local machine. If a machine will communicate repeatedly with a remote machine via the same channel, it is more efficient for application code to reuse the same connector for each transaction. This avoids unnecessary allocation of a new send-side event source and the required *drivers* to enable the actual communication. All code examples in this thesis use convenience methods to hide the complexity of the connectors mechanism at the cost of some runtime efficiency. As an example, this is the convenience method to send data to an `OverwritingBuffer` without explicitly using a connector:

[9]Unidirectional
[10]Bidirectional

```
//In OverwritingBuffer.java
public void send(T datum) {
  if (cachedSendConnector==null) cachedSendConnector =
      newConnector();
  cachedSendConnector.send(datum);
}
```

4.4.1.1 Sends to Non-destructive Channels

Destructive channels (such as `OverwritingBuffer`) are particularly easy to operate in an application as they can always have data sent to them. Non-destructive channels require much more care as Machine Java does not provide blocking send operations; sends are always non-blocking. To make programming with non-destructive channels slightly less onerous, every non-blocking connector has a single item send-buffer. This ensures that the first send operation on a connector can always return immediately without an issue. The data will be sent as soon as the receive end of the channel can accept it. Once the channel has accepted the data the non-blocking connector will register an event in the sender machine, and another send operation can be performed on the connector. If a connector is used again before the previous data item has been accepted by the remote channel, the send operation will throw a `PreviousDatumNotAcceptedException`. This means that code which will interact with non-destructive channels must always be sure to send data slower than it will be consumed at the remote end, or use an event-driven style to rate limit send operations. For example, to repeatedly send incrementing integers to another machine (`div2`) via a non-blocking channel (`input`), the following style of code must be used:

```
div2.input.send(nextSend, new Handler<Nothing>() {
  public void handle(Nothing info) {
    div2.input.send(++nextSend, this);
  }
});
```

In this example the `Nothing`-typed event handler is triggered at some point in the future after the first send operation. When the handler is executed it just sends again using the same event handler object. This causes an infinite stream of send operations and sender-side events, but the sends are perfectly limited to the rate at which the receiver can process the data. The more obvious non-event driven construction will throw a `PreviousDatumNotAcceptedException` very quickly:

```
while (true) {
  div2.input.send(++nextSend);
}
```

4.4.1.2 Intermachine Type Safety

Static type safety is ensured for channels through the use of Java generic types, just as it is for requesting new machines. Unidirectional channels have a single type parameter (an example can be seen on line 7 of figure 4.7), whereas bidirectional channels have a type for both the query and response directions. The send methods for channels are generic and only accept the configured type, and the event handler for a specific channel must also match the type of the channel. Type safety can be guaranteed across machines by the standard Java compiler because Machine Java uses instances of the application's machine classes (machine objects) to act as machine references. This allows the Java compiler to treat a distributed Machine Java application exactly as an ordinary non-distributed Java application.

For a machine, *a*, to be able to communicate with another machine, *b*, the application code in machine *a* must have a reference to *b*. The static type (the type of the variable, parameter or field as declared in the Java code) of the reference to machine *b* determines what fields are known to exist in *b*, so if the reference to *b* has the static type `Machine` then *a* cannot ever send data as the `Machine` class does not contain any channel declarations. If the static type of *b* in *a*'s code contains channel definitions then *a* can use these fields to access the channel object, which in turn enables communications with the live instance of *b*. The Java compiler enforces that input code is consistent with respect to the static types of machine references, so a reference to a `BasicReceiver` machine (shown in figure 4.3) cannot be assigned to a variable with a static type of `BasicSender`; this would cause a compile time error. The Java compiler also emits checks to ensure type safety at runtime, so even if an application uses convoluted code to circumvent static type safety, an exception will be thrown at runtime.

```
BasicSender wontWork
  = (BasicSender)(Machine)newMachine(BasicReceiver.class);
```

In this example (using the machine classes defined in figure 4.3) the compiler will accept the double cast as both casts are acceptable individually, but it will emit a runtime

check[11] to validate that the object is actually an instance of `BasicSender`. When executed this code will throw a `ClassCastException`.

Although machine references are subject to very strong type checks by the Java compiler and runtime, the data items accepted by a channel are not so rigorously enforced. This is because Java's generic types are *erased* at compile time; all instances of the same generic class contain the same runtime code regardless of the type parameter used in the code. This means that the compiler cannot automatically generate runtime class-cast checks to validate runtime type safety, and therefore it is possible to write code which will send data of the wrong type through a channel:

```
BasicReceiver rx = newMachine(BasicReceiver.class);
((Slot<String>)(Slot<?>)rx.numbers).send("Hello");
```

This example will not only compile, but it will also execute:
"`BasicReceiver@1,0:1> Hello`" is logged from the `BasicReceiver` machine. The Java compiler does detect that this code may be incorrect and will issue a cryptic warning:

Type safety: Unchecked cast from Slot<capture#1-of ?> to Slot<String>

This type of generic type safety violation can allow any sendable (see 4.4.1.3) type to be sent but it still does not violate the type safety of the receiver machine. In this case the code executes correctly because the `BasicReceiver` machine just invokes `Machine.log()` on whatever data arrives from the channel, and `log` accepts any object as a parameter. When non-trivial applications attempt to make use of a received data item as the type that the channel was declared to receive, the Java compiler will emit the necessary runtime checks. If the `BasicReceiver` class were modified to log the square of the newly received integer, like so:

```
public void handle(Envelope<Integer> info) {
  log(info.getPayload()*info.getPayload());
}
```

then a runtime exception is thrown:

java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer

If this `ClassCastException` is uncaught in the event handler then the handler will terminate abruptly and the next event in the machine's queue will be serviced. The

[11] Using a `checkcast` JVM bytecode.

overall operation of a machine is unaffected if an event handler terminates abruptly, but the application will not necessarily continue to function correctly if that handler provided important functionality.

To summarise Machine Java's intermachine type safety:

- It is impossible to write applications that use a non-existent channel in a machine reference. Most errors of this kind will be caught by the compiler, the rest will be caught at runtime before the channel is used.
- It *is* possible to write code that cannot make use of all the channels in a machine. This happens in the cases where the static type of a machine reference is not specific enough. However, this is a useful property as it allows object oriented 'implementation hiding' patterns to be applied in a machine oriented context.
- It is possible to to violate the generic type of a channel and send data of an unexpected type. This will very likely cause a `ClassCastException` to be thrown in the receiver machine during its channel event handler.

4.4.1.3 Sendable Objects: Immutable Types

Machine Java channels are only allowed to transfer *immutable* objects between machines. The application model does allow mutable data to be communicated where it is clear that a copy has been made, but this does not result in the most intuitive programming model. Restricting communication to only immutable data types most clearly conveys the concept of runtime intermachine isolation.

Unfortunately Java does not provide a general mechanism to express a requirement for a data type to be immutable; mutability of data is not encoded via the type system or standard annotations. There have been a number of proposals to extend Java to allow the expression of immutability, including in JSR[12] 308[63] and the work of Zibin et al[234]. These works include indications of immutability for data types, and to allow variables, parameters and fields to be constrained to only immutable data types. It does not seem likely that true support for immutability will arrive soon into Java as it is a considerable divergence from Java's typical mode of operation.

[12] *Java Specification Requests* (JSR) are part of the Java Community Process (JCP). See `https://jcp.org/en/jsr/all`.

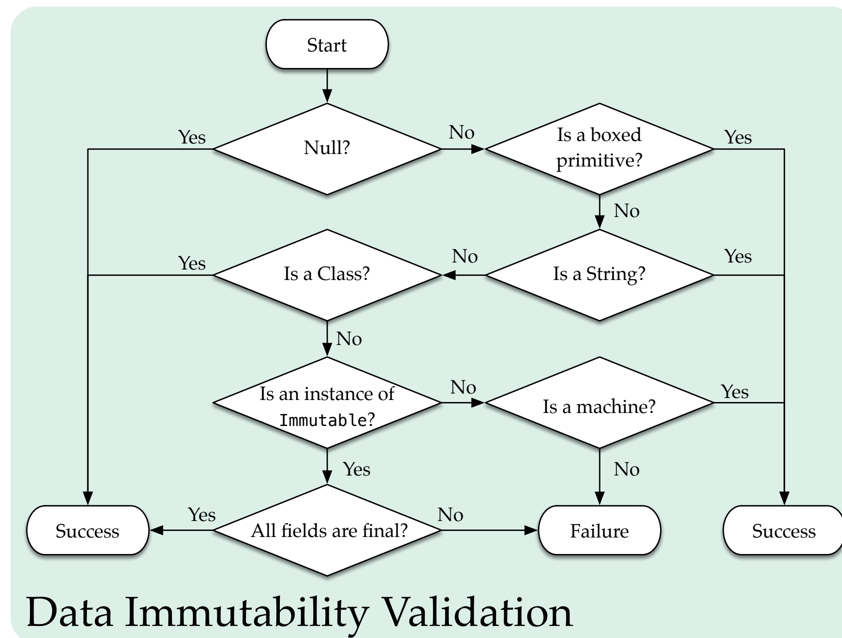


Figure 4.10: *The procedure for determining if an object is sufficiently immutable to be considered for intermachine communication.*

As Java does not provide a mechanism to statically guarantee the immutability of a data type, a more ad-hoc approach is required for Machine Java: The generic type parameter accepted for channel declarations is unconstrained; a channel can be defined to accept *any* Java object type. If a channel is defined for a mutable data type, such as an array (which is always mutable in Java), a runtime exception will be thrown on an attempt to send the mutable data type. The procedure to determine if an object is immutable can be seen in figure 4.10. Machine Java considers data to be sufficiently immutable for communication if it is one of: **`null`**[13], a boxed primitive (such as `java.lang.Boolean`, or `java.lang.Integer`), a `String`, a Java `Class<?>` object, a machine object, or an instance of Machine Java's `Immutable` interface.

Machine Java's `Immutable` interface can be implemented by a class to indicate that it is read-only after construction. Classes that implement `Immutable` must obey only a *weak* form of immutability where each field of the class must be **`final`**[14] to prevent changing the field's value after construction. The immutability of the field types is *not* enforced in Machine Java, but should be performed on a best-effort basis by the application programmer to avoid unexpected runtime behaviour. *Strong* immutable classes

[13] Although **`null`** values are not accepted by channels.
[14] In Java, **`final`** fields must be written to exactly once during the construction of the declaring object.

in Java require a number of conditions to be satisfied:

- Have only `final` fields.
- If the class is nested it must be a `static` class, or the enclosing class must also be immutable. This implies that *inner* (non-static nested) and *local* (non-static block scoped) classes can only be immutable if their enclosing class is also immutable.
- All accessible (`public`) fields must be primitive or immutable.
- Inaccessible (`private`) fields can be mutable (such as arrays), while the following sub-conditions hold:
  - The mutable field must be deep copied (referenced immutable objects do not need to be deep copied too) when the object is constructed.
  - The mutable object referenced by the field must be deep copied each time its reference would *escape* the class. A reference escapes when it is returned from an accessible method in the class, or when it is stored in another object.
- The class itself must be `final` to prevent subclasses from violating the above rules for accessing mutable fields.

The difficulty of implementing these requirements and of designing a verifier to enforce the requirements motivates Machine Java's considerably relaxed definition of immutable. Machine Java *does* guarantee no two machines will ever share a reference to mutable data, so it is not required that the programmer expend so much effort expressing this each time.

Machine Java's `Immutable` interface extends the `Flattenable` interface (see 4.7.3.4) provided by Network-Chi. This ensures that all immutable classes can also be serialised (*flattened*) inexpensively[15] at runtime.

4.4.2 Implementing Channel Protocols

Before the implementation of channels in Machine Java is discussed in the next section, it is helpful to briefly cover *internal drivers*.

[15]In this context inexpensive means avoiding dependence on full Java reflection, rather than a claim to efficiency.

4.4.2.1 Internal Drivers

Machine Java's application-facing APIs are able to function via abstract *drivers* where different implementations can be provided according to the needs of the specific platform. Drivers are only intended for use by Machine Java's event sources and internal code; applications should not use drivers directly. Machine Java defines five abstract drivers:

machine drivers (`MachineDrivers`) provides the execution contexts for machine classes and acts as the gateway between code executing in a machine context and other Machine Java framework functionality. Machine drivers provide the following important functionality:

- Provide *event managers* which isolate event sources from the implementation specific details of a machine's event queue. Event managers allow events to be registered and provide `IESIdentifiers`.
- Allow access to the machine's unique runtime identity, represented by the `MachineIdentifier` class.
- Provide instances of alarm and communications drivers to event sources. As these are provided on a per-machine basis each machine instance can be provided a different driver implementation if necessary.

alarm drivers (`AlarmDrivers`) can schedule code to be executed at some defined time in the future.

TPIF transmit drivers (`TPIFDriverTx`) can send messages to matching receive drivers.

TPIF receive drivers (`TPIFDriverRx`) receive messages sent to them by a matching transmit driver.

Processor drivers (`ProcessorDrivers`) coordinate the startup of each processor in the platform and are used by the **ProcessorManager** machine to create new machine drivers on the current processor.

4.4.2.2 Using TPIF Drivers

Channel implementations are able to communicate with other machines through the use of two abstract communications drivers: `TPIFDriverTx` to send data, and `TPIFDriverRx`

```java
package mjava.core.runtime.drivers;
import mjava.core.identifiers.IESIdentifier;

public abstract class TPIFDriverTx<T> {

  public TPIFDriverTx(IESIdentifier identity, boolean destructiverx,
      boolean buffered, Runnable accepted) {...}

  //Sends the specified datum to the matching RX end driver.
  public abstract void send(T datum) throws
     PreviousDatumNotAcceptedException;

  //Shuts down this driver so its resources can be reclaimed.
  public abstract void shutdown();
}
```

Figure 4.11: *An abridged version of the **TPIFDriverTx** abstract driver specification. See section 4.4.1.1 for a discussion of `PreviousDatumNotAcceptedException`.*

to receive data. These two drivers provide a *universal* message passing interface, which can be combined and configured in different ways to implement any of the the protocols discussed in the application model. To illustrate how channels are constructed in Machine Java the **OverwritingBuffer** protocol will be considered as it is the least complex non-trivial[16] protocol. **OverwritingBuffer** protocols do not have handshakes between sender and receiver, there is no return path, and the receiver can have any buffer size without impact on senders.

Abridged versions of the `TPIFDriverTx` and `TPIFDriverRx` Java classes can be seen in figures 4.11 and 4.12, respectively.

All code in the receive side of a channel implementation executes on the processor that the channel-defining machine is allocated to, with most of the code also executing in the context of the defining machine. The receive side of an **OverwritingBuffer** can be implemented as follows:

1. The `TPIFUnidirectionalProtocol<T>` (see figure 4.8) is subclassed. This marks the new class as `Intermachine` so it is permitted to be a **`public`** field in a machine, and it also designates the new class as an event source with the parameter type `Envelope<T>`.

[16]The **Constant** channel protocol is certainly less complex but it is trivial to the point of futility.

```
1  package mjava.core.runtime.drivers;
2
3  import mjava.core.Gettable;
4  import mjava.core.identifiers.IESIdentifier;
5
6  public abstract class TPIFDriverRx<T> implements Gettable<T> {
7    //Note: the Gettable interface defines the method:
8    //public T get();
9    //to retrieve the next datum, or null if none is available
10
11   public TPIFDriverRx(IESIdentifier identity, Runnable rxnotify, boolean
         destructive, int bufferLength) {
12     this.identity = identity;
13     this.rxnotify = rxnotify;
14     this.destructive = destructive;
15     this.bufferLength = bufferLength;
16   }
17
18   public int getBufferLength() {...}
19
20   //Allows this driver to invoke the rxnotify Runnable when the next
        datum is available.
21   public abstract void accept();
22
23   // Shuts down this driver so its resources can be reclaimed.
24   public abstract void shutdown();
25
26   //Gets an item from this driver and blocks until one arrives none are
        available.
27   public abstract T getBlocking();
28 }
```

Figure 4.12: *An abridged version of the **TPIFDriverRx** abstract driver specification. Note that the ability to perform a non-blocking* `get()` *is provided by its implementation of the* `Gettable` *<T> interface.*

2. In its constructor the new channel class must request a `TPIFDriverRx` driver from the machine that owns the new channel instance. It can accomplish this by:

```
owner.getMachineDriver().getTPIFDriverRx(...)}
```

A receive driver requires four parameters to determine its behaviour:

an address This driver will receive messages from senders with a matching address. This is represented by an *intermachine event source identifier* (a `IESIdentifier` object). `EventSource` provides functionality to obtain this identifier for the channel instance.

buffer size This determines how much memory the implementation will allocate to buffers, and also changes the behaviour of the driver when the buffer size is zero.

send destructivity The destructivity of send operations. If the driver is a non-destructive type then handshaking is used between sender and receiver drivers to prevent buffer overflow conditions in the receiver. In this case a *destructive* driver is requested. Destructive receive buffers must be buffered for event driven operation as without buffering any received data items would be lost by the time an event handler could be executed to retrieve the data. Unbuffered destructive protocols can be expressed with `TPIFDriverRx`, but this requires a channel to use blocking receive operations.

interrupt handler Finally, the `TPIFDriverRx` instance requires an *interrupt handler*. This is a Java `Runnable` that will be invoked when the driver has data ready to be read from its buffer. This interrupt handler can be executed in any context without synchronisation to application code, therefore it is critical that it does not use any data structures that are also used by code executed in an application context. For an **OverwritingBuffer** implementation this interrupt handler just registers an event:

```
registerEvent(new TPIFEnvelope<T>(_drv, OverwritingBuffer.this));
```

where `_drv` is a reference to the receive driver instance, and `OverwritingBuffer.this` is the channel instance.

The data is not read from the receive driver during the interrupt handler,

instead the `Envelope`[17] object will read it from the buffer during the application code's event handler. This is an important design pattern across Machine Java's channel implementations, as it ensures that data is consumed from buffers at a rate dictated by the application and not the implementation of the channel protocol. The receive driver will not invoke the interrupt handler again until the driver's `accept()` method is called to indicate that this is allowed. The `Envelope` class invokes the driver's `accept()` immediately after application code uses `getPayload()` to retrieve the latest datum. This mechanism ensures that the channel implementation is compliant with the application model's event queue specification that at most one event is queued for a specific event source at any instant.

From an application's perspective an `Envelope` object is just a container for the new data item which in some cases can provide additional information or facilities; bidirectional channels use `ReturnableEnvelopes` which both provide a reference to the querying machine (with a `Machine` static type), and also provide a `reply(R data)` method to directly respond to the querying machine. However, envelope classes are quite active entities and only provide the illusion of data encapsulation through just-in-time reads and data caching.

3. After the receive driver has been obtained from the machine driver, the `accept()` method is invoked on the driver to enable notification for the first data item. The receive side of the channel is now complete and will behave as expected.

The send side of an **OverwritingBuffer** channel is somewhat less complex as it is not an event source. Naturally, all code in the send side of a channel executes in the context of the machine with data to send and this means that implementations must be especially careful not to attempt to violate machine isolation: Channel classes contain code that can be executed in more than one machine context and therefore the data stored by channel objects will have apparently different values depending on the particular machine that is executing the code.

1. A new subclass of `TPIFDestructiveConnector<T>` is made to represent connec-

[17] TPIFEnvelope<T> is an implementation of the `Envelope<T>` interface, which has only one method: `getPayload()`

tors for this channel type.

2. The new connector class must request a send driver in its constructor. This is very similar to how the receive side requests its driver, but because the send-side code executes in a machine that does not *own* the channel, it must request the send driver from the currently executing machine:

```
1 Machine.getThisMachine().getMachineDriver().getTPIFDriverRx(...)}
```

Again there are four parameters that determine a send driver's behaviour:

send address The `IESIdentifier` of the receive driver that should receive sent messages. This is straightforward for an overwriting buffer implementation as only the receive side is an event source so there is not any confusion about which identifier to use. The send side code can just use the channel object's inherited `getExternalID()` method. The implementation of the current machine driver will ensure that the ID returned for the send side code will match the ID provided on the receive side.

Bidirectional and non-destructive connectors are themselves event sources in the send side machine so must take care to use the identifier of the channel and not the connector.

destructive send This must match the value used for the receive driver as it determines the handshaking behaviour of the driver. Destructive send drivers do not have to ask permission from the receiver to send data.

buffering Specifies if the receiver uses a send buffer or not. This only affects the operation of non-destructive driver pairs. When a sender driver is in non-destructive and unbuffered mode all send operations become blocking until the receive driver is ready to receive. This is the avenue to implementation of truly synchronised rendezvous protocols. In the case of this destructive **OverwritingBuffer** protocol the value does not matter.

interrupt handler Send drivers can invoke an interrupt handler when they are ready to send more data. Destructive protocols, such as this one, are always ready and this handler is never invoked. For non-destructive protocols this interrupt handler is invoked after the most recently 'sent' data item has actually been transmitted to the receive driver. Just as with the receiver driver, the send driver's interrupt handler can be invoked in any execution context

and cannot safely share data structures with the machine that created the connector instance.

3. Finally, a convenience method (such as the one seen on page 155) can be provided to avoid application code having to manage channel connectors.

All of the remaining unidirectional protocols can be implemented using this style of construction, and the bidirectional protocols can be implemented with multiple sender and receiver drivers. Every instance of a receive driver implies some resource consumption in its host processor, even if the driver is unbuffered or used for void-valued channels. This is because the lower level implementations must be able to determine which driver instance is responsible for handling data items that arrive to its communications resources. This is unlikely to present an issue for unidirectional channels as a machine class can only define a finite number of fields to be populated by channels, and these are fixed at compile-time. From an engineering perspective resource consumption is less troublesome if it is known when the system is designed, but this is not necessarily the case for bidirectional protocols. Therefore bidirectional channels are more problematic as their connectors must have their own ability to receive data (implying a receive driver), and a machine can instantiate an unbounded number of connectors. This is especially problematic when the convenience method pattern is used to query a remote channel, such as in this example[18]:

```
1 srv.replyService.query(new PingMsg(), new PingReplyHandler());
```

where `replyService` is a bidirectional `RemoteProcedureCall` channel:

```
1 public final RemoteProcedureCall<PingMsg, PingMsg> replyService = ...
```

In this example, a new connector could be created each time `query()` is invoked which would lead to a memory leak in the implementation. Machine Java avoids memory leaks in this situation by using the concept of *ephemeral* connectors. These connectors can only be used once, as their receive drivers are closed when their data is retrieved from the associated `Envelope` object by the connector's event handler.

For unidirectional protocols the connectors used by convenience methods are cached; only one connector will ever exist for each channel of each machine reference owned.

[18]These code excerpts are taken from the *SpeedTest* micro-benchmark. The full source code for this benchmark is provided in appendix section C.1

```
package mjava.core.runtime.drivers;

public abstract class AlarmDriver {

  public AlarmDriver(Runnable handler) {...}

  //Sets the alarm time for this driver.
  public abstract void setAlarm(long activateTime);

  //This will cancel the alarm.
  public abstract void cancel();
}
```

Figure 4.13: *An abridged version of the **AlarmDriver** abstract driver specification.*

This avoids unexpected proliferation of the implicit single-item send buffers within non-destructive connectors.

A complete reproduction of a **RemoteProcedureCall** channel implementation in Machine Java can be found in appendix A.1.

4.4.3 Spontaneous Event Sources

All four of the model described spontaneous event sources (see 3.2.3.2) are easily expressed and have already been implemented in Machine Java. The time bound event sources (**alarms**, **delays**, and **periodics**) are all implementable using Machine Java's abstract `AlarmDriver`. The alarm driver is very simple in comparison to the send and receive drivers, and its contract can be seen in figure 4.13. As with the communications drivers, the alarm driver uses a pseudo-interrupt to asynchronously notify timing event sources that the specified time has occurred, and the interrupt handler has the same execution caveat too: It cannot safely access data structures shared with application code as it can be executed from *any* execution context. An alarm driver can only have one alarm pending at once and does not need to be explicitly shutdown to free its resources; a finished or cancelled alarm uses no resources and is eligible for garbage collection when unreferenced. `Yield` event sources do not require any drivers to operate; the functionality provided by extending the `EventSource` abstract class is sufficient to implement a **Yield**-type event source.

The hierarchy of spontaneous event sources can be seen in figure 4.14. Spontaneous event sources are particularly uncomplicated to use from within application code as the

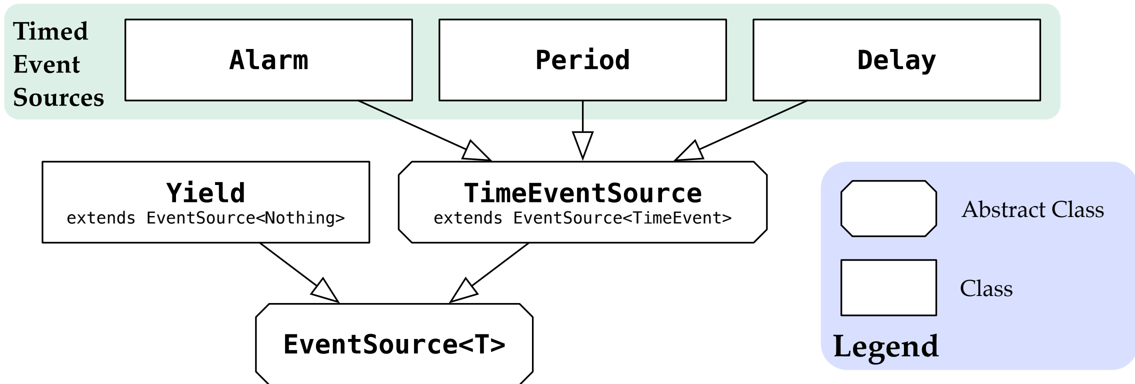


Figure 4.14: *A summary of the spontaneous event source library provided in Machine Java.*

timed event sources only require a time specification parameter and an event handler, and `Yields` only require an event handler. In accordance with the application model, Machine Java's event handlers are non-preemptive so in the following periodic event example:[19]

```
new Period(1_000, new Handler<TimeEvent>() {
  public void handle(TimeEvent info) {
    doWork();
  }
});
```

the `doWork()` method is guaranteed to execute no more frequently than once every second but actually is not guaranteed to execute at all when other event handlers cause interference. All absolute time specifications in Machine Java are relative to the same 'zero-point' time as Java's `System.currentTimeMillis()` time stamp method. This means that the following code will register an event in one second time on all platforms, regardless of their (possibly non-compliant) implementation of `currentTimeMillis()`:

```
new Alarm(System.currentTimeMillis()+1_000,
          new Handler<TimeEvent>(){ ... });
```

A complete reproduction of a **period** timed event source implementation in Machine Java can be found in appendix A.2.

4.4.4 Processor Managers

Machine Java's realisation of MAA framework **ProcessorManager** machines have extended responsibility beyond that specified in the model. Not only do they respond to

[19] These code examples use an underscore (_) character to group the digits of the millisecond constants. Underscore characters are allowed in numeric literals from Java 7 onwards and have no effect on the value represented by the literal.

```
1 // The ProcessorManager is the only implicit machine on a processor.
2 @Relax
3 public final class ProcessorManager extends Machine {
4
5   //Processor-local requests:
6   public <T extends Machine> T requestMachine(Class<T> typeToStart,
        final Handler<T> machineArrivedHandler, boolean blocking) {...}
7
8
9   //Remote request box
10  public final RemoteProcedureCall<MachineControl,
        MachineControlResponse> remoteRequest = new RemoteProcedureCall
        <MachineControl, MachineControlResponse>(new
        RemoteRequestHandler());
11
12
13  //Only for debugging and instrumentation!
14  public final RemoteDataFetch<ProcessorStatus> status = new
        RemoteDataFetch<ProcessorStatus>(new Handler<ReturnableEnvelope
        <Nothing, ProcessorStatus>>() {...});
15 }
```

Figure 4.15: *An abridged version of the* ***ProcessorManager*** *class definition. Application code never interacts with* ***ProcessorManager*** *instances directly; they are used indirectly via the* `Machine` *class.*

requests for new machines just as described in the model, but **ProcessorManager** classes also provide perform *dynamic* allocation of machines to processors. Machine Java applications are unaware of the processor manager concept as it is hidden by `Machine`'s methods to request new machines.

The Java definition of the **ProcessorManager** can be found in figure 4.15. It can be seen that **ProcessorManager** is annotated with `@Relax` which is necessary as it declares a public method: `requestMachine`. This method is used by `Machine` to implement the `newMachine` family of methods. `Machine.newMachine()` is implemented as follows:

```
1 public final <T extends Machine> T newMachine(Class<T> type) {
2   return Platform.getPlatform().getThisProcessorDriver().
        getProcessorManager().requestMachine(type, null, true);
3 }
```

Although the implementation is circuitous, indirection of machine requests via the local **ProcessorManager** substantially reduces application complexity as all allocation decisions and error handing can be managed by the framework. The Platform API (discussed in section 4.5) can only suggest which processors might be *statically* suitable

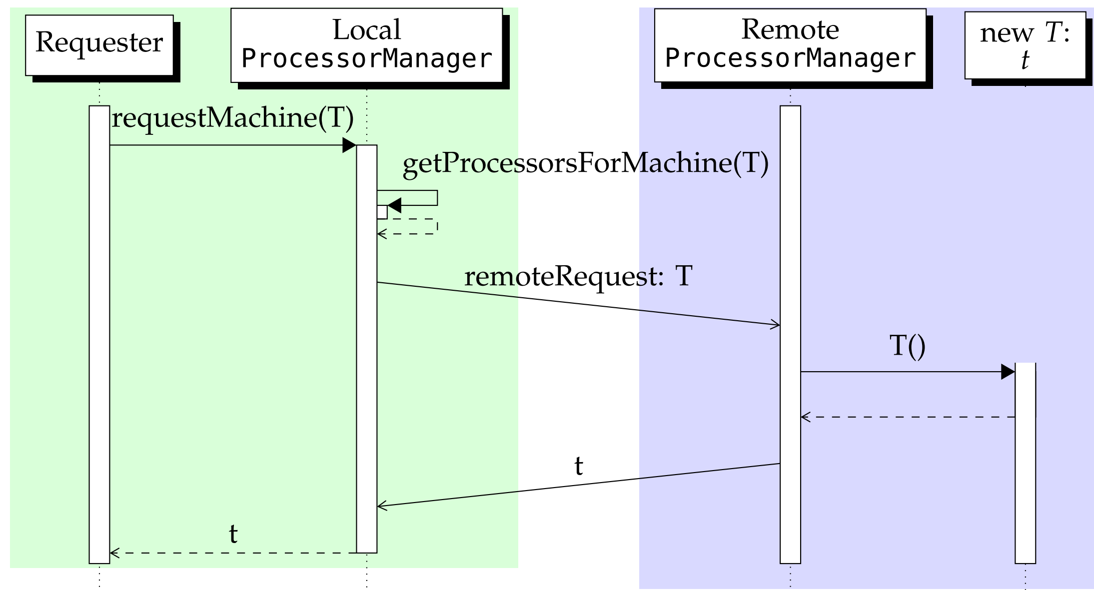


Figure 4.16: *A sequence diagram of the request for a new machine in an event handler in 'Requester'. The processor manager on the same processor handles the request by determining which remote processor's* ***ProcessorManager*** *to forward the request to.*

for a given machine type but not a single specific processor. Therefore without the indirection through **ProcessorManager**'s dynamic allocator, every machine would have to implement its own. The overall sequence for a blocking machine request can be seen in figure 4.16, but event handlers have been elided into the machines for simplicity. The sequence of events is as follows:

1. An event handler in the requesting machine ('requester') uses the blocking method `Machine.newMachine()` to request a new machine of type T.

2. `newMachine()` uses the internal API to get a reference to the local **ProcessorManager** and then invokes `requestMachine()` on the `ProcesssorManager`.

3. The **ProcessorManager** uses the platform API to determine the possible processors for a new machine of type T. This is a fast operation that uses no remote resources.

4. The **ProcessorManager** picks a processor according to its dynamic allocation algorithm, discussed below. It then uses the platform API again to get a reference to the remote **ProcessorManager** located on the picked processor.

5. The local **ProcessorManager**, which is still executing in the context of 'requester', issues a blocking query to the 'remoteRequest' channel of the remote manager. The query contains a `MachineControl` describing the intent to create a machine and

the required machine type. The 'requestor' machine is now stalled until this query receives a response, but the local **ProcessorManager** can still service requests as its execution context has not been stalled.

6. The remote **ProcessorManager** services the request at some point in the future, subject to the delay in servicing the its outstanding queued events. When the request's event handler executes the manager attempts to create a machine by requesting a new machine from the processor's `ProcessorDriver`. The driver will perform the necessary operations to create a new execution context and create a new instance of T, causing T's constructor to be executed.

7. When the new instance of T: t has completed construction successfully, a reference to t is sent as the reply to the query inside a `MachineControlResponse` message.

8. The requestor's **ProcessorManager** resumes execution when the reference to t arrives. As it was a blocking query, there is no event for this data arrival, the blocked execution context will resume as soon as the implementation allows.

9. The `newMachine` method call returns the new reference to t. If the request for a new T had failed, `newMachine` will throw a runtime exception with a message specified by the remote **ProcessorManager**.

As figure 4.16 shows, only the constructor of the new machine is synchronous with the request. All future event handlers in the new machine, t, happen asynchronously with respect to other machines.

The **ProcessorManager**s dynamic machine allocator is only a basic proof of concept and follows the following operating principle:

1. The **ProcessorManager** has an internal 'pick counter', n , which is initialised to zero when the manager is created.

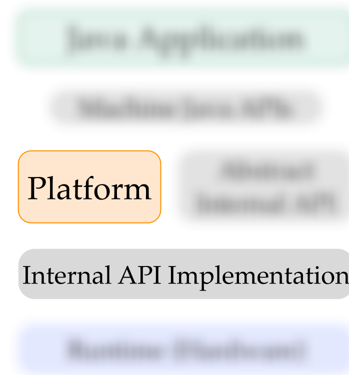
2. The platform's `getProcessorsForMachine()` API is used to get an iterator for the potential processors. If this iterator has no items then a runtime exception is thrown to indicate that the machine cannot be constructed anywhere.

3. If the iterator has fewer items than n, then n is reset to 0. Otherwise n is incremented.

4. The processor at the index n is selected.

Although simple and inefficient, this procedure ensures that a **ProcessorManager** will eventually attempt to use every processor offered by the platform API. Future implementations could easily extend the allocator to enable intelligent load-balancing decisions based on acquired knowledge. For example, more sophisticated `ProcessorManagers` could maintain an active but low priority stream of exchanges with other close by processors. If these messages include performance or loading data then this would allow `ProcessorManagers` to make better informed decisions about the allocation of machines long after system initialisation.

4.5 Platform API



This section discusses Machine Java's platform API and its relationship to an internal API implementation.

Machine Java provides a library of abstract classes to enable the representation of *platforms* in Java code. Both static structural representations of a platform and more dynamic styles of platform description can be accommodated using this approach. Modelling a platform in Java has the additional advantage that the runtime platform API (discussed in section 3.3.5) can be implemented directly in Java code rather than relying on a tool-flow for generation. Machine Java platform realisations are initially much closer to the *fully dynamic* and *hybrid static-dynamic* approaches, as each processor in the platform can use the same platform implementation and have its behaviour determined at runtime. However this is not a requirement of the Machine Java architecture and it would be possible for each processor to have its own separate platform implementation on the condition that all of the 'views' of the platform are consistent.

The lowest level components in Machine Java's stack (shown in full in figure 4.1 on page 128) address different platform-specific concerns in a system:

Platform *instances* are constructed using the platform API by a hardware designer or system integrator. A platform instance describes the structure and capabilities of a hardware architecture. Platform instances reference a specific internal API implementation that can enable Machine Java to execute on it. Platform descriptions are covered in this section using the *XYNetwork* platform as a reference example. Code defined in platform instances is not intended to interact directly with hardware.

Internal API implementations must meet the contracts of the internal API for a specific processor's hardware architecture. An internal API implementation can be re-used by multiple platform instances, and a heterogenous platform instance may reference multiple different API implementations for its various processor types. The 'networkchi' internal API implementation is covered in section 4.6. The internal API implementation is the only Machine Java component that should interact

```java
package mjava.core.runtime;

//The platform class describes the overall system architecture.
public abstract class Platform {
  //Helpers for code on this processor:
  public abstract Processor getThisProcessor();
  public abstract ProcessorDriver getThisProcessorDriver();
  public MachineDriver<?> locateMachineDriver() {...}
  //Access to remote processors:
  public abstract Iterable<Processor> getProcessorsForMachine(Class
      <? extends Machine> type);

  public abstract ProcessorManager getProcessorManager(Processor
      target);
  public abstract Processor getProcessor(int procId);

  //trigger Machine Java framework startup:
  public abstract void startProcessor(Class<? extends Start>
      startMachine);

  //'gateway' to the active platform:
  public static Platform getPlatform() {...}
}
```

Figure 4.17: *An abridged version of the* ***Platform*** *abstract class. Instances of this class define which internal API implementation Machine Java will use, how the application can be mapped to the available processors, and provide the ability to startup the Machine Java framework.*

directly with the underlying hardware.

Hardware is assumed to exist and all salient details have been captured by the platform and internal API implementations. All other hardware details including but not limited to design, manufacture, FPGA configuration, processor bootloading, and flash programming are out of scope for Machine Java.

Machine Java's platform API broadly follows the platform model described in section 3.3: A platform is a set of processors with defined resources and abilities to communicate with other processors. The platform API is composed of just four abstract classes: `Platform`, `Processor`, `Resource`, and `Communication`.

4.5.1 Platform Classes

The platform class is the core of a platform description. Its specification can be seen in figure 4.17. This is where all important functionality is described; the other platform API classes are simple representations of their counterpart in the platform model. The *active* platform for the current processor can always be obtained in framework code by using the static `Platform.getPlatform()` method highlighted in figure 4.17. The platform object provided by the `getPlatform()` method is only required to supply information relevant to the current processor, and what is considered relevant is the decision of the platform implementation. Information about the global structure of the platform is only available if a specific implementation makes it available. Application code and high-level framework implementations are intended to be platform independent and so should never directly use any functionality provided by Machine Java's platform API.

The `Platform` class provides three broad categories of functionality: getting internal API implementations, working with other local processors, and startup functionality.

internal API access The most important driver that a platform instance can supply is the machine driver for the current execution context (via `locateMachineDriver()`). This facility is ultimately how non-machine framework code is able to request drivers from the current machine, as the `Machine` class's static `getThisMachine()` method is internally implemented as:

```
1 public static Machine getThisMachine() {
2     return Platform
3       .getPlatform()
4       .locateMachineDriver()
5       .getMachine();
6 }
```

Platform instances can also provide access to the local processor's driver via `getThisProcessorDriver`. Processor drivers are responsible for coordinating the creation and maintenance of execution contexts for machine drivers, and depending on the implementation the processor driver may also have responsibility for managing low-level details such as the initialisation of the local processor, and the multiplexing of communications drivers with available hardware resources.

local processors The platform also enables interaction with other local processors. 'Open-ended' communication between processors is not supported in Machine Java, it can only happen between machines and then only from a channel connector to a

channel. If another machine is already known it can be communicated with via its channels. As described in the model's framework section (3.4), even requesting new machines from other processors follows the same communications model: a request is sent via an ordinary channel to the **ProcessorManager** of the remote machine.

Obtaining a reference to a remote processor manager is a two stage process:

1. First a `Processor` object for the remote processor must be obtained. These can be provided if the processor's identity (represented by a Java integer) is known which is the case if a message has ever been received from that remote processor. If the remote processor has never been contacted before, then the only path to obtaining a `Processor` object for it is via the `getProcessorsForMachine()` method. This method provides a Java `Iterable`[20] of processors that are *statically* capable of hosting the machine class specified. The iterator provides processors in some statically defined order of preference from most desirable to least desirable. As these preferences and capabilities are statically determined, and machine oriented applications can have dynamic structures, there is no guarantee that a processor supplied by `getProcessorsForMachine()` will accept a request for a new machine. There is also no guarantee that a 'more preferable' processor as returned by this method will actually be better at runtime according to any criteria. The static determinations are just the best guesses of the compiler, platform implementation, or both.
2. Next, the Platform API enables a **ProcessorManager** machine reference to be retrieved for a given processor. This is done via the `getProcessorManager()` platform method, which is also how the current processor's **ProcessorManager** is obtained.

The definition of 'locality' for other processors is the concern of the platform instance, so another processor is considered local if under any circumstances the platform offers it for use.

framework startup Finally, the platform instance provides the 'entry point' to the Machine Java framework. The `startProcessor()` method is invoked by the `Start.`

[20]The `Iterable<T>` interface provide access to an `Iterator<T>` and enables the implementing class to be used in Java's `for (Object x : iterableAggregate)` statement.

```
1  package mjava.core.systems;
2
3  public abstract class Processor {
4
5    public abstract String getName();
6    public abstract String shortDescription();
7
8    public abstract int getID();
9
10   //Only works if:
11   //Platform.getPlatform().getThisProcessor().equals(this)
12   public abstract List<? extends Communication>
       getCommunicationsResources();
13   public abstract int hasResource(Resource query);
14 }
```

Figure 4.18: *An abridged version of the* ***Processor*** *abstract class.*

`start()` static method which is itself called by the application's Java entry point. `startProcessor()` is responsible for the creation of the local processor driver and this will then create the **ProcessorManager** for the current processor. Implementations that do not use Java's **`public static void`** `main()` method as an entry point must ensure by other means that `startProcessor()` is invoked, such as in the platform class' static initialiser.

As processors can be identified by Java integer, this implies that each processor is limited to around two billion local processors with which it can communicate. In the unlikely event that this particular construction of the Machine Abstract Architecture survives long enough for only two billion *local* processors to be stifling, contemporary systems engineers may consider extending the processor identifier to a **`long`**.

4.5.2 Processors

`Processor` classes provide very little functionality to a system, but their general contract can be seen in figure 4.18.

A processor object for the current processor is able to enumerate the available communications resources but platform implementations do not have to be able to enumerate the communications resources of remote processors. In the same way, only the current processor object must be able to provide sensible responses to the `hasResource()`

method. Processor objects for remote processors can throw exceptions for these methods or return an empty value. The `hasResource()` method is used to determine the number of a particular resources the processor has regardless of their status as used or available. Processor drivers can use this to validate the construction of additional machine that require resources. The `getName()` and `shortDescription()` are to facilitate better debugging and pretty printing, but serve no functional purpose. Application and framework code must not depend on the output of these methods.

4.5.3 Resources

`Resource` classes are purely symbolic; they define no members. Subclasses of `Resource` are created for each type of resource that can exist and each instance represents a particular instance of an exclusively allocatable runtime resource.

Differences between processors can be modelled using this resources abstraction. A machine that can benefit from execution on a 'fast' processor could require a 'fast processor' resource that is only present (and possibly in limited quantities) on the most capable platform processors. Attempts to allocate this machine to a slow processor would then fail.

4.5.4 Communications

The `Communication` abstract class extends `Resource` and allows any communications device that can reach another processor to be represented. Its functionality is also very limited:

```
1 public abstract class Communication extends Resource {
2   public abstract boolean isProcessorReachable(Processor p);
3 }
```

As with the methods defined in `Processor`, the `isProcessorReachable()` method is only required to operate as expected if the communication resource is owned by the current processor. This method enables communications driver implementations to determine which local communications resource is appropriate to use to send data a remote processor. Machine Java's platform API does not attempt to capture the relative value of communications resources.

4.5.5 The XYNetwork Platform

To evaluate the Machine Java framework it is necessary to consider a variety of hardware configurations. *XYNetwork* is a flexible platform implementation that can enable Machine Java applications to execute on any homogeneous, two-dimensional, regular rectangular mesh of processors, including real hardware meshes and simulated meshes in a JVM. This platform is also adequate for the exploitation of symmetric multiprocessor hardware, such as standard PCs, where it is less important to account for the physical arrangement of the processors.

The XYNetwork platform is intended to be the simplest possible platform that can support a nearly unbounded number of processors. The number of processors is only limited by its use of Java integers to represent the absolute X and Y coordinates of each processor in the mesh. Integers and absolute addresses are used for purely practical reasons. The XYNetwork could be extended to *relative* addressing with arbitrary precision integers[21] without invalidating its theory of operation. In this construction all processors would consider themselves to be at coordinates $(0, 0)$ which would require another mechanism to determine if the current processor is also the platform's 'first' processor.

This platform design is not sophisticated and makes a number of simplifying assumptions about supported hardware targets:

- All processors are identical in all ways except their network coordinates. This assumption has two important implications:
  - All processors have access to the same executable code. It is immaterial whether the code is replicated for each processor or somehow shared between them.
  - All processors have the same resources.
- Every processor has a single communications resource to represent the mesh network, and all other processors are reachable via this resource.
- The executable code contains implementations of all possible machine types in the application code; any processor can be used to host any machine type.

[21] Such as Java's `java.math.BigInteger` class.

- The proximity of two processors has some impact on the performance of communications between them, and possibly some impact on the communications performance of other unrelated processors. That is, it is assumed to be preferable for communicating processors to be close to one another.

- A practical assumption (but theoretically unimportant) is that the hardware is supported by the 'networkchi' implementation of Machine Java's internal APIs. This implementation is covered in section 4.6.

As the XYPlatform can only represent homogeneous systems, resource constraints are not considered and each processor is modelled as having access to one of every possible resource; its `hasResource()` method returns 1 regardless of the supplied parameter. Additionally, the XYPlatform is unable to make use of any facts about the current application's structure; there is no way to use compiler or programmer supplied hints about the best runtime allocation of machines. This means that the platform's response to the `getProcessorsForMachine()` method can only be the result of a processor-proximity directed heuristic as processors can only be differentiated by their location in the mesh.

4.5.5.1 Allocating Machines to Processors

Altogether these assumptions lead to a short implementation: There is only one necessary processor class which represents a processor at a specific set of coordinates. This processor's (`XYProcessor`) list of communications resources is a single item: `XYNetworkInterface`, and this resource's `isProcessorReachable()` method returns **`true`** for all other processors. The `XYNetworkPlatform` class itself constructs a processor driver from the 'networkchi' implementation during the `startProcessor()` method and this driver eventually provides all runtime behaviour for the current processor.

So far this platform definition could be applied to any homogeneous, fully-connected network topology and not just cartesian meshes: only the network address stored in the processor representation would need to be altered. The implementation of the `getProcessorsForMachine()` method is what binds a platform to a specific topology.

A platform's `getProcessorsForMachine()` could only be topology-agnostic if a human or compiler provided it with a static set of processors that are referred to in every instance; it cannot create a new `Processor` instance without some understanding of the hardware's addressing scheme. The ability of `getProcessorsForMachine()` to order

the returned processors also depends on an understanding of the nature of the hardware's communications interconnect. In general it makes little sense to have a topology agnostic platform implementation as platforms are intended to represent the execution substrate meaningfully.

The most effective `getProcessorsForMachine()` implementations will certainly take advantage of application specific knowledge. However, optimising the allocation of machines (representing units of work) between processors depends on both the requirements of the machine types and on the system-specific non-functional requirements. Examples of two different machine allocation strategies are shown in figures 4.19 and 4.20.

Within the constraints imposed by XYPlatform's assumptions there are very limited options for the implementation of `getProcessorsForMachine()`. It cannot be application specific and it must be the same algorithm on every processor. Non-functional requirements and runtime variability cannot be addressed by such a simple allocation strategy. Previous work has established that toolchain-based approaches (such as the Machine Java framework in this case) are well suited to deal with the challenges of runtime variability[76].

The current processor's **ProcessorManager** always has ultimate discretion about where a new machine will be allocated but as it is platform independent and cannot compare processors according to any static criteria. The **ProcessorManager** can only decide a processor's suitability through active communication with the remote processor manager located on it.

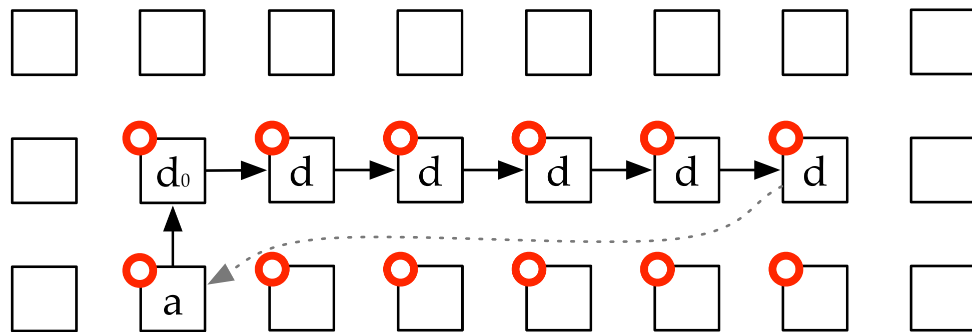


Figure 4.19: *The allocation of a six-stage d-type machine pipeline where* $(x+1, y)$ *is chosen if the requesting machine is also a d. Router usage assumes simple Y then X routing.*

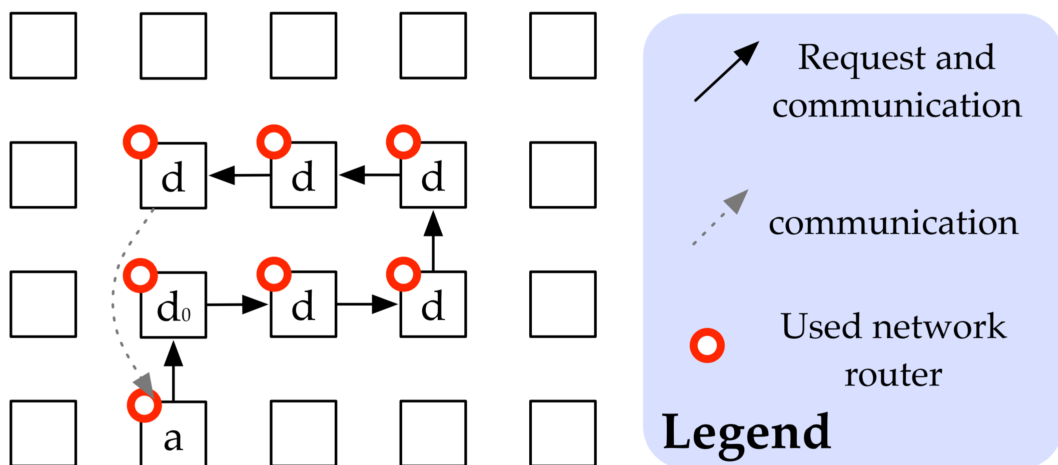


Figure 4.20: *The allocation of a six-stage d-type machine pipeline where the next processor is a non-trivial function of current processor coordinates, the acceptable folded pipeline dimensions, and the coordinates of* d_0*. Although the end of the pipeline may be closer to the start, this may not improve the desired non-functional characteristics.*

4.6 Implementation in Standard Java

Fundamentally, the Machine Java framework is a standard Java 7 application and for this reason the majority of its construction is uninteresting.

However, there are a number of non-obvious techniques which enable Machine Java's non-standard behaviour to be realised in a standard JVM. These include:

- How the framework starts.
- How machine instances are identified.
- How machines are created.
- How machine isolation is guaranteed.
- How a channel object is able to communicate with a remote machine.

This reference implementation of Machine Java only has one implementation of its internal APIs and this 'networkchi' driver package depends heavily on the Network-Chi compiler to support its operation. Network-Chi and its construction is discussed more thoroughly in section 4.7.

4.6.1 Starting the Machine Java Framework

When executing in a JVM, a Machine Java application begins its execution from the `public static void main()` method in the class specified to the JVM as the application's entry point. From this point the following sequence happens:

1. Application code invokes `Start.start(...)` with the class of the application's *start* machine.
2. The `Start` class requests the active platform and instructs it to 'startup' the platform processor that is executing the code: `Platform.getPlatform().startProcessor (startClass);`
3. The platform instance creates a new processor driver, and instructs it to begin operation with the start machine specified by the application's entry point.
4. The processor driver starts by creating a 'timing nexus' to manage processor local timed event sources, and a 'communications nexus' which multiplexes all

machine-oriented communications too and from Network-Chi's mesh networking abstraction. On construction the communications nexus registers an interrupt handler with the network-API enabling the Machine Java framework to receive asynchronous notifications for new received network messages.

5. A new **ProcessorManager** machine is created for the local processor.

6. If the processor driver is executing on the platform's *first* processor then it will also create an instance of the application's start machine, via the local **ProcessorManager**:

```
if (Network.amOrigin()) {
  manager.newProcessorLocalMachine(startMachineType);
}
```

   The `Network` class is Network-Chi's mesh networking API discussed in section 4.7.3.2. `Network.amOrigin()` returns true if the current node's coordinates are $(0,0)$, and this runtime defines the origin node to be the first processor of the platform too.

7. As `amOrigin()` is the first network API used, Network-Chi initialises the network runtime:

   (a) If the framework has not yet been initialised then this means it must be the first time this code has been invoked and therefore this code is executing on the notional origin node. The following initialisation begins:

   (b) For each processor in Network-Chi's configured mesh network size, a new thread is created.

   (c) Each new thread begins executing the original `main()` method from step 1. The entry point is determined by walking the current call-stack to find the first instance of a static main method. Reflection is used to invoke this method.

   (d) Another new thread is created for each node to receive network messages via UDP.

   (e) The initialisation of the 'network' is now complete.

8. The processor driver now enters its event loop. The startup events for the **ProcessorManager** and the application's start machine will be in the event queue already.

9. The timing nexus is allowed an opportunity to consider if anything will happen soon.

10. The next event is fetched from the processor's priority queue if there is one. If there is, this is serviced in the context of the machine that issued the event.

11. The loop begins again from step 9.

4.6.2 Machine Instance Identification

Internally, Machine Java uses a `MachineIdentifier` class to uniquely identify all machines at runtime. Application code is never intended to use `MachineIdentifier` objects; application code should use proper machine references (Java references to machine objects). The `MachineIdentifier` class is a triple of the platform-defined processor identification for the processor that hosts the machine, a Java `Class` object representing the machine's type, and finally a **`long`** which represents the machine's 'serial number' on its host machine. The information contained in a `MachineIdentifier` is sufficient to construct new machine objects of the correct class that can reference the *active* instance of the machine.

4.6.3 Machine Creation and Channel Addressing

The creation of a new active machine is an intricate processes as it also determines the 'addresses' of all the channels in the new machine instance. Starting from the point where the processor's local **ProcessorManager** has made the decision to construct a new active machine, the following happens:

1. The **ProcessorManager** requests the local processor driver creates a new *active* machine:

```
T m = Platform.getPlatform()
      .getThisProcessorDriver()
      .constructActiveMachine(type);
```

2. The processor driver creates a new identity for the new machine, and then creates a new `ActiveMachineDriver` instance. This machine driver represents the execution context of the new machine and in alternative implementations could contain its own thread and its own event queue.

3. The processor driver then switches into the context of the new machine which does not yet have an object.

4. The processor driver then uses Network-Chi's `EssentialReflection` library to create a new instance of the machine type:
   `public static <T extends Machine> T createMachineDynamic(Class<T> mClass )`. On a JVM this method uses standard Java reflection to perform a number of basic checks that the machine type adheres to machine Java's rules (see section 4.3.3). An instance of the machine class is then created using its nullary constructor.

5. The constructor defined in the `Machine` superclass begins to execute:

   (a) The new machine requests its driver from the platform:
   `_driver = Platform.getPlatform().locateMachineDriver();`

   (b) The new machine provides the Machine Java framework with an *early* reference to its object:
   `_driver.machineObjectHasArrived(this, new InternalExecutor());`
   This serves two purposes: code used by the machine once its constructor begins to execute may invoke `public static Machine getThisMachine()` which depends on the framework having a reference to the new machine. The second purpose is to supply the machine's driver with a runnable that can invoke the machine's `protected internal()` method. This method would not be visible to framework code as the application's `internal()` method will be declared in another package. This avoids the requirement on another use of Java reflection which is particularly expensive to implement.

6. The machine class' constructor now begins to execute:

   (a) First any initialised fields are populated by executing their initialiser. All channels defined by the machine are declared **`final`** so they must have an initialiser at this stage or the explicit constructor (the next step) must initialise them. When a channel is constructed:

      i. The constructor in `EventSource` begins to execute as this is the root ancestor of all event sources.

ii. The *owner* machine is recorded in the `EventSource`:

```
public final Machine owner = Machine.getThisMachine();
```

This is why it was critical for `Machine`'s constructor to provide the framework with a reference to the machine object so early on.

iii. The `EventSource` constructor gets an event manager implementation from the owner machine. The event manager implementation is subsequently used to supply *intermachine event source identifiers* (`IESIdentifier`) to the channel's implementation.

iv. The channel's specific constructor begins to execute.

v. The channel requests a new `IESIdentifier` to uniquely identify this channel instance, enabling remote machines to communicate with it. The important detail is that the `IESIdentifiers` are well defined and are issued in sequence. For a machine being constructed with a particular `MachineIdentifier` the sequence of `IESIdentifiers` issued will always be the same.

(b) The application's defined constructor in the machine class (if there is one) now executes.

(c) All channels have now been initialised, along with any other application specific initialisation. As Java has a well-defined order in which class fields are initialised[74, §12.5], the `IESIdentifiers` issued to each channel are also well-defined.

(d) The application defined constructor now finishes and control returns to the processor driver.

7. The processor driver reverts the execution context and adds the new machine's startup event onto the processor's event queue.

8. A reference to the new machine object is obtained from the new machine's driver and returned to the **ProcessorManager**.

4.6.4 Machine Isolation and Communication

Machine isolation is guaranteed[22] as machines never have true references to one another, even if they are executing on the same processor. When a machine has a reference to another machine, the reference is a valid and fully constructed instance of the correct machine class but it is *never* the same object as the original object constructed when the machine was first created on its host processor.

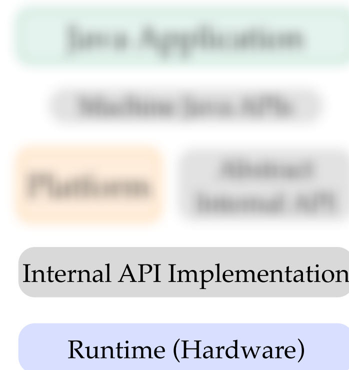
The machine objects used as references are *shadow* machines. These have the same class, have been initialised identically but use a `ShadowMachineDriver` rather than the `ActiveMachineDriver` described previously. `ShadowMachineDriver`s do not execute their `internal()` method and do not pass events through to the processor's event queue. During the initialisation of a shadow machine the execution context masquerades as the original machine that will be referenced. The `IESIdentifiers` supplied to the channel instances are therefore identical to the original active machine's and are used by the channel implementations to create valid connector instances. Code that requests drivers from the framework during a shadow machine's initialisation will receive inert but compatible driver objects that do not use any resources. Resource assertions (`REQUIRE_RESOURCE` and `ENSURE_SINGLETON`) are ignored for shadow machines.

This isolation mechanism is extremely robust as it cannot even be broken by reflection: application code simply does not have a reference to the correct object to reflect upon. Circumventing this isolation is only possible if the active machine is present on the same processor as the attempt to break isolation, and the process is extremely convoluted requiring many layers of Machine Java's framework to be painstakingly traversed. Intermachine isolation is not intended to be secure against a determined attack on a JVM, but it does prevent even sophisticated attempts to 'work around' the framework's restrictions. Where security is a true concern, disabling reflection for application code via Java's security manager mechanism would be sufficient to guarantee machine isolation.

[22] except for abuse of **`static`** fields, which cannot easily be restricted in standard Java

4.7 Implementation on Bare-Metal Hardware

In this chapter the MAA application model has been applied to the Java programming language. This yields an expressive and scalable programming framework suitable for massive multiprocessing, and in particular multiprocessors without an expectation of shared memory between processing elements. Within the bounds of a system the machine oriented model also avoids a number assumptions on potentially expensive runtime capabilities, including:



This section addresses the compilation of Machine Java applications to platforms without standard Java or operating systems.

preemptive multitasking as machines are internally non-preemptive and intermachine relationships are unknowable within an application; The application model is tolerant of preemptive and non-preemptive mappings of machines to processors as machines do not inherently supply timing guarantees.

network implementation is abstracted by channels, so no particular protocol (such as TCP/IP) or networking strategy is assumed by an application.

memory protection is a static property of a Java application, so no additional operating system or hardware memory protection is required by the machine model.

threading is not provided to applications; concurrency is facilitated via machines. This is one of the many divergences from standard Java.

filesystems (and graphical user interfaces, and other input/output interfaces) are not assumed by the Machine Java framework.

The freedom from assumptions only applies within the context of a system. At the boundaries of a system, where the environment or other systems will be interacted with, then use of some of these facilities (such as networking or IO) is inevitable. The overall principle is that hardware assumptions are only introduced by application code, and these assumptions may be spatially confined to the machine that makes them. For

example, the `BasicSender` machine on page 136 sends a single integer value to a single-buffered channel in another singleton machine. A sufficiently sophisticated implementation *could* provide the same semantics in silicon by latching a constant into a register. The application model does not imply any other capabilities or behaviour for this machine. However, there is a wide gulf between the smallest possible implementation of a machine's written semantics (in program code or transistors), and what is feasible to automatically implement. A machine's ultimate implementation is not *guaranteed* to be compact by the model, it is just not guaranteed to be large either.

The implementation of Machine Java in a resource constrained context presents a substantial challenge:

- Java is a substantially complex programming language with a large runtime.
- Versions of Java aimed at embedded uses-cases are still too large or impose unacceptable caveats on the language, such as limited libraries or incompatible definitions of standard types.
- Machine Java is highly dynamic: machine objects are created indirectly without the use of the **`new`** operator, and objects are transparently communicated between processors of potentially different architectures.
- Machine Java does not necessarily require a number of Java features including threading, object monitors and concurrency libraries. In addition applications are unlikely to use most of Java's libraries, especially those that do not make sense in a resource constrained context. It is desirable to avoid the memory cost associated with unused or useless Java libraries.

Ordinarily, standard Java is compiled to an extremely compact and high-level bytecode but this requires a Java Virtual Machine[122] (JVM) and a large set of libraries[23], together often called a Java Runtime Environment (JRE), for execution. However, the size of the standard Java libraries is prohibitive in the context of resource constrained processors.

[23] With Java 1.7.0_79-b15 on MacOS 10.10.4 the *rt.jar* for the runtime is 62MiB.

Section Overview

To address these challenges, this section discusses a strategy for translating the Java programming language, and in particular the subset required by Machine Java applications, to a form that is suitable for execution on resource limited embedded systems such as limited processors in Networks-on-Chip, softcore processors in FPGAs, and microcontrollers. The translation strategy prioritises the minimisation of runtime memory usage, generated code size, and suitability for a wide range of limited architectures over other desirable goals such as execution speed and strict adherence to the Java standard.

The translation procedure, or *Concrete Hardware Implementation*[24] (Chi) of a software application first converts the application's compiled Java class files to a self-contained intermediate representation conducive to optimisation and refactoring. The intermediate format is then serialised into the C programming language which is then compiled to target-specific machine code via any C99[97] compliant compiler. This section covers the techniques for analysing whole Java applications (section 4.7.1.2), translating Java methods (section 4.7.1.4) and building a stand-alone application(section 4.7.2) with the same functional behaviour as the original Java. Through the use of aggressive code pruning and a very compact runtime memory organisation, minimal Java applications can be translated to standalone, bare-metal binaries that require less than 32KiB of program code and less than 8KiB of runtime heap.

Java's built-in networking abstractions (provided in `java.net`) are effective for IPC via a local area network or the internet, but are inappropriate for communications between processors in an on-chip network. Rather than expose low-level hardware details to Java applications, a simplified and compiler assisted networking API is provided to applications. The `Network` API (section 4.7.3) is a mesh based abstraction for network communication, allowing the programmer to send Java objects to other nodes without consideration for the underlying hardware topology or protocols. Owing to the tight integration of compiler and networking API, the combination is referred to as *Network-Chi*.

[24]The name reflects the concretisation of the Java input. The output binary does not contain or depend on a JVM and does not require an operating system for embedded targets.

4.7.1 Compilation Strategy

The general strategy for implementation of Java applications on embedded systems is the Ahead-of-Time (AOT) compilation and optimisation of Java bytecode to compilable C, and then to machine code via an existing C compiler for the target. This is in contrast to the Just-in-Time (JIT) compilation at runtime favoured by modern JVMs. The code generated by Chi does not depend on any external libraries or the presence of an operating system. To achieve this goal, three main tactics are applied to reduce the overhead of the Java environment. These are:

- Only including methods, objects and Java functionality that are used by the application under compilation. (Section 4.7.1.2)
- Identifying and re-implementing commonly used Java libraries that cause runaway inclusion of further classes. (Section 4.7.1.3)
- Abstraction of fundamental target system characteristics so that the code-generator can be reused for different processor architectures.

The compiled application behaves exactly as it would have done inside a JVM, but without any virtualisation or access to the JRE's libraries. This is accomplished via AOT linking of required class files rather than the standard demand-loading of referenced classes.

4.7.1.1 Machine Java to Bare-Metal Workflow

A high-level overview of the workflow necessary to execute a Machine Java application on a network is shown in figure 4.21 with the main steps from application to hardware being:

1. **Java compilation** The application is compiled with any compliant Java 7 compiler[25] using the Network-Chi libraries packaged into the same Java `.jar` file as the extended compiler. As this stage of development is the same as building any Java application with external libraries, developers can retain their familiar IDEs and design tools. The output of this stage is a collection of Java **`.class`** files.

[25] Java 8 compilers should also work acceptably as Java 8 did not introduce any modifications to the JVM. The new features in the Java 8 language (such as functional interfaces) should also work correctly in the workflow described, but as this is not important to the thesis it has not been verified.

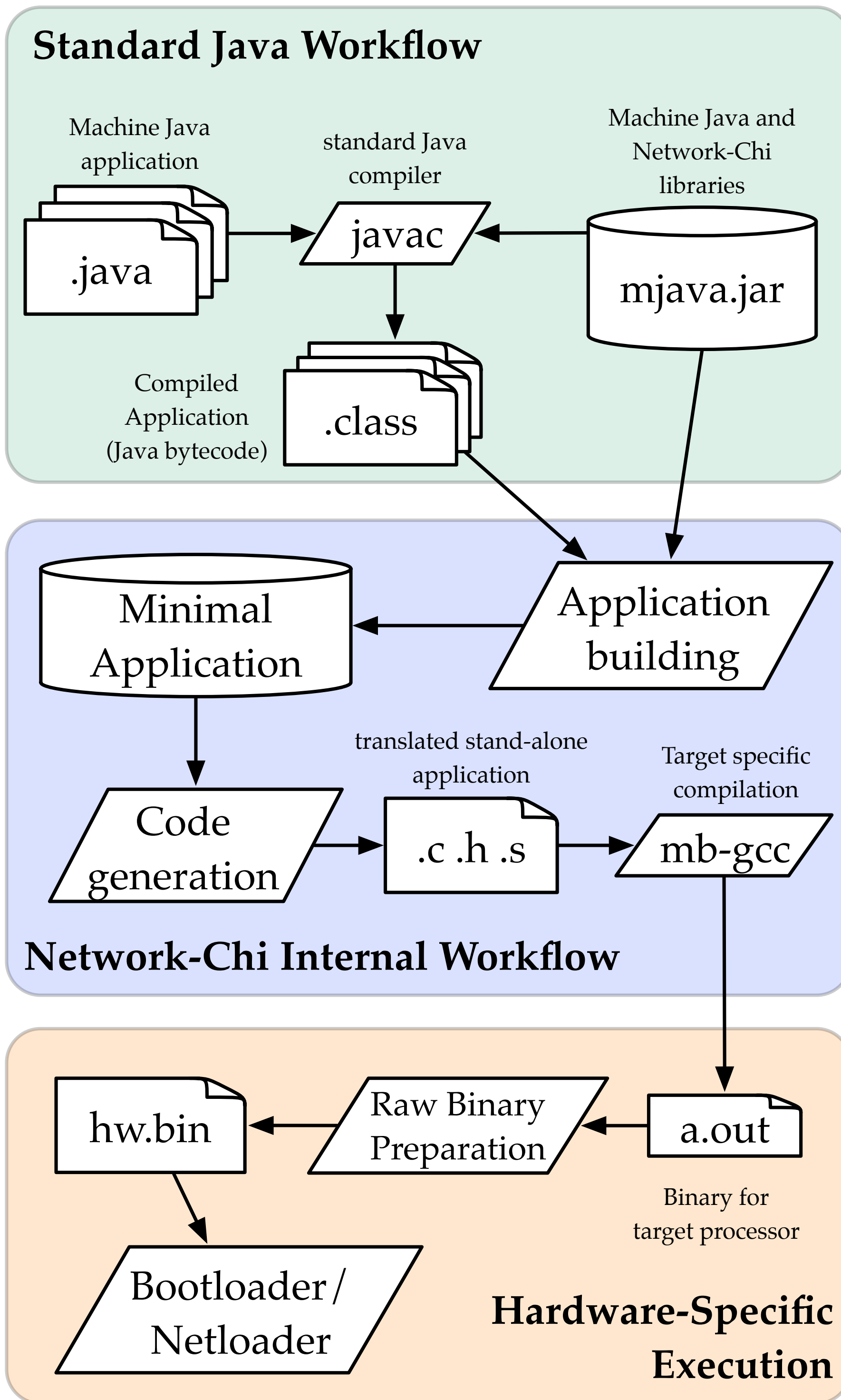


Figure 4.21: *The workflow to build a Machine Java application using Network-Chi: The standard Java compiler is used to create* `.class` *files for the application. The compiler then* explores *the application to find used classes, generates standard C, and automatically compiles the C using the appropriate downstream compiler.*

2. **Chi is executed on the class files** The Chi tool is invoked on the `.class` files, instructed about the target architecture, and what level of Java safety to retain in the output. Considerable reductions in binary size and improvements in runtime performance can be achieved by not including Java's runtime checks. However the safety of the resulting binary will be compromised if the application is not functionally correct, or if the application is structured to depend on an exception (such as a `ArrayIndexOutOfBoundsException`) being thrown by the Java runtime. The main internal steps of the Chi compiler are:

    (a) **Application building** An internal representation of the whole input application is built. The internal model of the application only includes reachable code and used classes. Each method used in the application is converted from JVM byte code instructions into an internal single-assignment representation that is more conducive to transformation and code generation. This procedure is considered in more detail in section 4.7.1.2.

    (b) **Intermediate code generation** The internal application is serialised into a subset of standard C99[97] code that implements the original semantics of the input application but without a dependancy on a JVM. This code can be compiled with any C99 compiler, (including some with only partial support, such as GCC[67]) ensuring wide compatibility across embedded processors and a high degree of optimisation in the final binary code. A number of commercial and research tools [14, 7, 212] also use C as an intermediate language for Java compilation because of these compelling advantages.

    (c) **Target binary building** The appropriate build tools for the C code are invoked automatically. Linker scripts are automatically generated to ensure code and data are placed into the correct NoC memory areas, automatic generation allows for Chi to rearrange memory sections depending on configured stack and heap sizes.

3. **The binary is prepared and bootloaded** The build tools will have emitted a binary in a format that depends on both the platform and the target compiler, this must be converted into a format suitable for use by the NoC bootloader and then sent to the NoC for execution. The precise nature of this process is out of scope for this paper, as is building the NoC itself.

As Chi must support multiple different target architectures the differences between these platforms are abstracted by a `ComputationalModel` class. The model describes to the code generator the differentiating characteristics of the target platform including native data types, endianness, alignments, available memory and peripherals. The model also describes the capabilities of the target such as support for floating point arithmetic, recursion and dynamic method dispatch.

These models are used during the application building procedure to verify that the input application is compatible with the capabilities of the target architecture. For example, if the model describes an architecture that would prefer to only statically dispatch method invocations then the application building procedure will issue an error if the application under compilation requires dynamic dispatch to be behaviourally consistent with standard Java.

4.7.1.2 Application Building

Application building is the first stage of the Chi translation procedure where standard Java **`.class`** files are used to build a complete internal representation of the user's application in memory. This internal representation includes all code that is reachable from the entry point of the user's application, including used code in the standard Java libraries if necessary.

The transformation process uses compiled Java classes to avoid the complexity associated with parsing Java source code and to take advantage of the sophisticated verification and optimisation that compliant Java compilers perform. Additionally, a considerable number of Java's language features are handled in the source compilation and are no longer present in the Java class files. Some of these features include: class nesting, generics, autoboxing, monitor handling for **`synchronized`** methods, and multidimentional array accesses.

The Java class file format[122] is very stable and has been changed far fewer times than the Java programming language itself. The Network-Chi tooling uses the *Apache Byte Code Engineering Library*[10] (BCEL) to access and interpret the necessary Java class files.

The purpose of application building is to aggregate enough information about the methods and classes used such that subsequent translation steps do not have to refer back to the class files. Application building gathers the following information about a

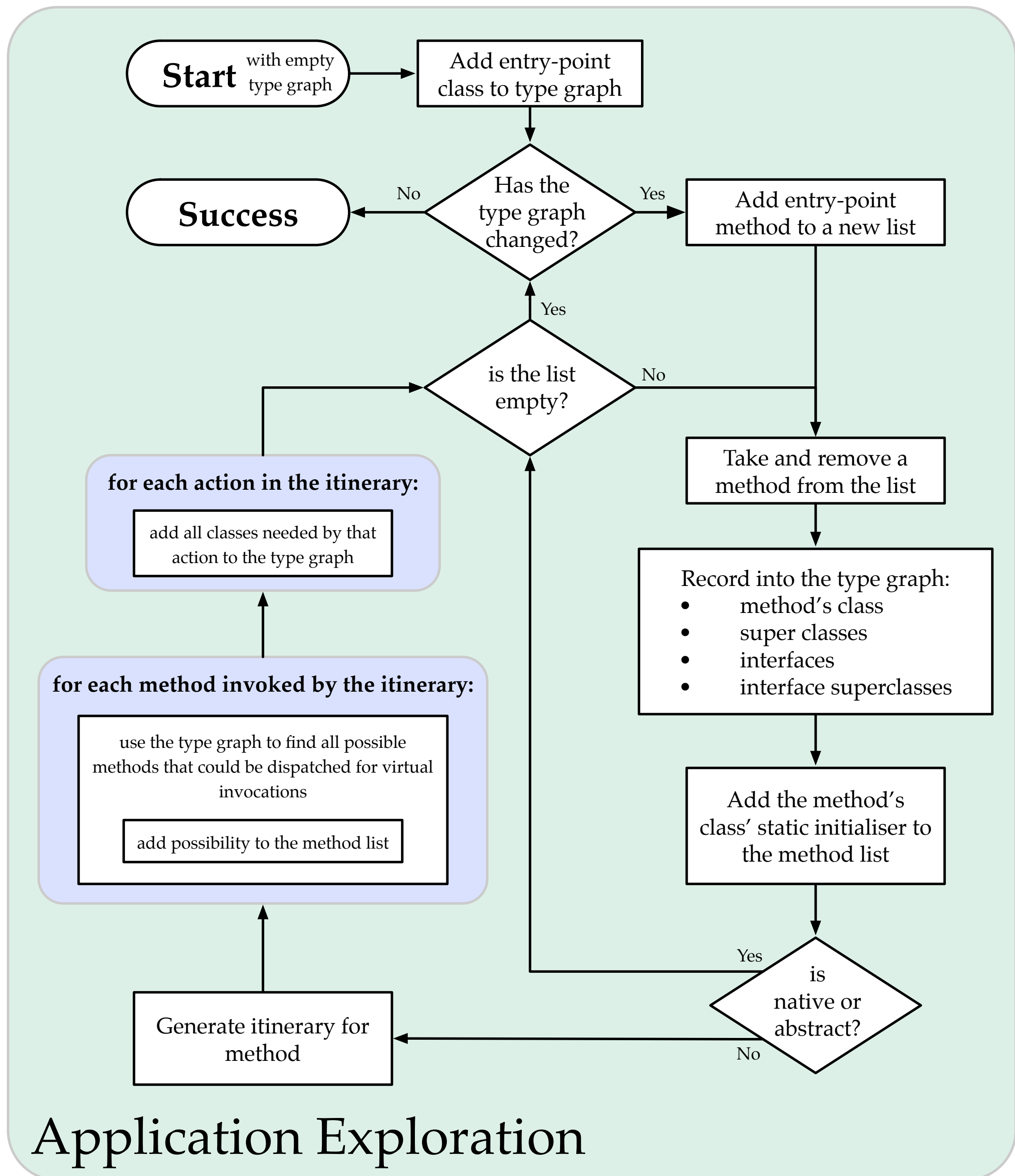


Figure 4.22: *The procedure for exploring a Java application starting with only it's entry point. Classes are retrieved on-demand using the same loading mechanism as the application would use at runtime.*

Java application:

- The entry point of the application.
- Every method callable by the application in a form called an *Itinerary* (see section 4.7.1.4). Itineraries describe the list of actions a method performs when invoked.
- Every class initialiser method (`<clinit>`) present from all classes used by the application. Class initialiser methods are used by Java to initialise default values for class static fields.
- The graph of all Java classes used by the application. This type graph considers interfaces to be superclasses of their implementations.
- The set of callable native methods. These cannot be automatically translated so subsequent code-generation will emit a warning and produce compatible stub code to allow compilation to continue.

The application building procedure is an iterative process that can be seen in Figure 4.22. This procedure is essentially a variant of Bacon's *Rapid Type Analysis*[21] algorithm for analysis of whole applications in statically typed languages. The primary, although minor, difference is that Chi does not require an enumerable Call Hierarchy Graph (CHG) nor a Program Virtual-call Graph (PVG) as these are produced during the exploration of the input application. Rather than attempt to enumerate all classes and methods and then prune the unused ones from the system, the internal application model is built by an explorative process of repeatedly discovering new used types and then considering if virtually dispatched methods now have a new choice of body for execution because of these discoveries. This has two main advantages: Firstly that only classes actually used by the application are incorporated into the model which minimises application translation times. Secondly, the discovery procedure is able to use the running JVM's own built-in class loaders. Using the JVM's class loaders to lookup classes and their bytecode allows the use of prepackaged and networked class repositories in the input application.

The drawback of this process is that it cannot work on applications where classes are used that are never referenced in the code itself. This can happen in two main ways: First, if a native method declares it returns an object of some type T, it is permitted to return any subclass of T and this subclass may never have been referenced

in any code reachable by the user's application. Secondly this situation arises if an `ObjectInputStream`[26] is used to read in a serialised object. The deserialized object could be of any class whatsoever, including ones never referenced by the application. Ordinary JVMs would attempt to lookup the class of the serialised object at runtime but this is not an option for Chi translated Java. However, these are not significant drawbacks in context as whole machine-oriented applications (or consistent partitions) are compiled at once. This means that an unknown object type could never be received via a channel as a machine under compilation must have sent it. In addition, general purpose Java serialisation is not supported and Java native methods are not compilable using this technique, so are not able to return unexpected, unresolvable types at runtime.

4.7.1.3 Replacement Classes and Methods

Method calls within an application to even a seemingly innocuous library method such as `System.out.println("Hello World!")` causes a vast number of classes[27] to become required due to transitive dependencies. This runaway in class dependencies is addressed by a framework for replacing classes and methods. Importantly, replacement classes and methods are completely transparent to the input Java application.

The active `ComputationalModel` class defines which classes and methods need to have alternative implementations in order for Java to target their platform. All targets are assumed to have both `java.lang.Object` (the eventual ancestor of all classes) and `java.lang.Class` replaced.

The standard implementations of `Object` and `Class` contain multiple native methods and where appropriate implementations of these are provided. Most functionality provided by `Class` is not supported in Chi, so the replaced methods are just minimal stubs. Where a feature *is* supported but the implementation is not expressible in standard Java the replacement method is annotated with `@ChiNative` to indicate that the code generator must provide the functionality itself rather than use the translated the body of the method. The method `java.lang.Object.hashCode()` is an example of such an annotated method. Using a new annotation (`@ChiNative`) rather than Java's

[26]Java's deserialisation helper stream.
[27]401 classes loaded: as determined by "`java -verbose mjava.tools.chi.tests.HelloWorld | grep Loaded | wc -l`" on Java 1.7.0_79-b15 on MacOS 10.10.4

`native` modifier allows a standard Java-compatible method body to be provided too. This is especially useful for functionality provided only by Network-Chi (such as the `Network` API, see section 4.7.3) which can have a meaningful ersatz implementation in a JVM.

Other classes in the set of default replacements include `String`, `System`, and `PrintStream`. The implementation of `String` in Java has an extraordinary number of dependencies rendering it far too large for small embedded systems. A considerably simpler (although without complete unicode support) replacement implementation is applied by default. In total only eighteen classes are in the default replacement set, of which seven are to replace the standard libraries boxed primitives. The other replacements only have minor changes to remove further library dependencies (such as on system properties or security).

4.7.1.4 Itinerary Generation

Within the Network-Chi tool method bodies are represented as objects called *itineraries* which describe a sequence of actions (derived from the original Java bytecodes) upon objects and registers. The itinerary representation of methods is somewhat different to the JVM model of computation. Itineraries do not use an operand stack to pass information between actions, and there is no notion of a program counter. Where a JVM instruction would receive its operands on the operand stack, the corresponding action in an itinerary will expect its operands in specific registers (of which there are an unlimited number), and it will place any result into another specific register. Likewise `Branch` actions do not have program counter addresses of their branch targets but symbolic references to the action that would execute next depending on the situation.

The procedure used to generate Chi itineraries from a Java method is illustrated in Figure 4.23.

Network-Chi uses 29 different action classes to represent the 201 valid JVM instructions. The reason for this reduction is that whole categories of JVM instructions become single actions (almost all arithmetic instructions are implemented by a `TwoOperandArithmetic` action) and all stack manipulation instructions are removed during itinerary building as there is no runtime operand stack in a Chi compiled application.

All actions in an itinerary contain information such as the exception handlers that will catch an exception they might happen to throw at runtime, and the line number in

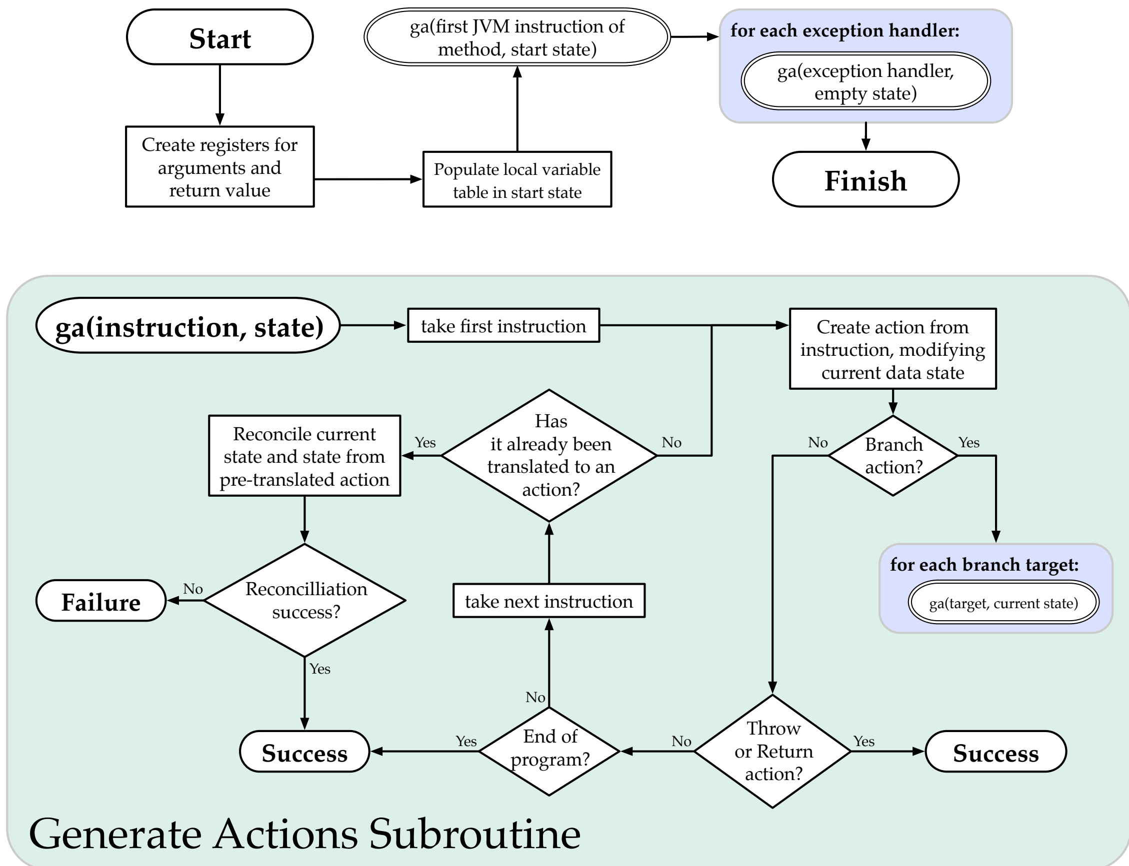


Figure 4.23: *The procedure for constructing a high-level 'itinerary' of actions from a Java method body. This procedure converts the stack-machine oriented Java bytecode into a more manageable single-assignment representation.*

the original Java source code that they were derived from.

Itinerary generation is used to gather the following information about a method:

- An ordered list of actions performed by the method.
- A list of input registers this itinerary will use to receive its arguments.
- An output register if the method is non-void to return the result.
- A set of all temporary registers used by the actions of this itinerary.

An example of the JVM bytecode for a simple method (`BasicSender.internal()` from figure 4.3), and its equivalent form as an itinerary action list can be seen in figure 4.24. The notation used in figure 4.24 does not capture all of the relevant details and is only meant to be indicative of an action list's paradigm; the text representation of an action list is not used within Chi and is only intended for human inspection.

During itinerary building a notional operand stack and local variable table is maintained in a structure called the Data State. The data state keeps a mapping from the JVM operand stack and local variables to the registers which now represent the storage in the itinerary. JVM instructions that would modify either local variables or the operand stack in fact make modifications to the data state instead. When an action is generated the data state is checked to ensure that it is consistent with expectations and when the control-flow analysis leads the itinerary generation procedure back to an instruction already translated, the current data state is reconciled with how the state was at the time when the instruction was first translated. Both local variable tables are compared and incompatible variables are marked as unreadable then both operand stacks are compared for consistency. If the stacks contain incompatible types in any identical positions or are different lengths then the bytecode does not meet JVM specifications and translation is aborted. This procedure is very similar to bytecode verification from the JVM specification[122].

Exception handlers appear to be unreachable as there is no branching control flow that is able to reach them in the bytecode, therefore they are explicitly translated to actions. If the active computational model has all exceptions disabled then exception handlers are indeed unreachable and they are removed as dead code.

```
protected void internal();
  Code:
     0: aload_0
     1: ldc #25//class examples/helloworld/BasicReceiver
     3: invokevirtual #27//Method newMachine:(Ljava/lang/Class;)Lmjava/core/Machine;
     6: checkcast #25//class examples/helloworld/BasicReceiver
     9: astore_1
    10: aload_1
    11: getfield #31//Field examples/helloworld/BasicReceiver.numbers:Lmjava/core/tpif/Slot;
    14: sipush 4919
    17: invokestatic #35//Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    20: invokevirtual #41//Method mjava/core/tpif/Slot.send:(Ljava/lang/Object;)V
    23: return
```

JVM bytecode for `BasicSender.internal()`

```
examples.helloworld.BasicSender: SRR0 <= this
SRR1 <= examples.helloworld.BasicReceiver
SRR2 <= Invoke Lexamples/helloworld/BasicSender;.newMachine(Lmjava/
    tools/chi/replacements/SimplerClass;)Lmjava/core/Machine; on object
    SRR0
Check SRR2 instanceof Lexamples/helloworld/BasicReceiver;
examples.helloworld.BasicReceiver: rx <= SRR2
examples.helloworld.BasicReceiver: SRR3 <= rx
SRR4 <= ((examples.helloworld.BasicReceiver)SRR3).numbers
SRR5 <= 4919
SRR6 <= Invoke static Lmjava/tools/chi/replacements/SimplerInteger;.
    valueOf(I)Lmjava/tools/chi/replacements/SimplerInteger;
Invoke Lmjava/core/tpif/Slot;.send(Lmjava/tools/chi/replacements/
    SimplerObject;)V on object SRR4
Return(void)
```

A Network-Chi itinerary for `BasicSender.internal()`

Figure 4.24: *The JVM bytecode for the* `BasicSender.internal()` *method seen in figure 4.3, and the itinerary that Chi generates for it. The eventual C code for this method is not included as it is over 200 lines long.*

4.7.2 Language Runtime

Network-Chi has two main runtime components: the language runtime and the network runtime. This section discusses the language runtime which encompasses the general execution strategy in C, the runtime memory representation of objects, memory management and the entry point to the application. The network runtime addresses the startup of each processor and configuration of necessary hardware, and the implementation of the `Network` library. The network runtime is covered in section 4.7.3.

4.7.2.1 In-Memory Representations

The runtime architecture of a Chi application is extremely simple compared to a standard JVM. All objects and arrays are stored in memory[28] and code generators are free to chose how to represent temporary registers. The internal arrangement of objects in memory is defined by the *Concrete Binary Object* (CBO) format. This format is used for all code generators and target architectures as the specific sizes of fields is dependent on the active computational model. The model provides helpers for encoding and decoding primitive types for its specific architecture. The in-memory layout of objects uses only one integer field (of a width defined by the active computational model) to represent the runtime type of the object, and then each of the object's fields, in alphabetical order grouped by the class they were declared in ancestor-first order (ie: `java.lang.Object`'s fields first). If the target architecture requires data alignment then padding fields are inserted into the CBO before each unaligned field. The CBO's main purpose is to allow internal compilation stages to manipulate the runtime representation of an object while maintaining application consistency, and also to allow compilation stages to create architecture-dependent object 'literals' in a flat binary representation. This can be used to prepopulate memory regions with valid objects without runtime code to allocate or construct the objects.

Arrays have one extra integer of overhead which is the number of items the array can store. This is placed after the type identifier and before the array data. It is safe for an object to omit an '`isArray`' field because array manipulation operations can only be used in Java on an array typed variable. Any time there is a narrowing cast of a reference in Java, such as an `Object` reference to a variable of type `Object[]` the compiler emits a

[28]They must have an address in memory, rather than have their value stored in registers.

`checkcast` instruction that would throw an exception if the cast is found to be invalid. Because of these guards it is always safe to assume that a reference points to the correct type of object, even in the case of arrays.

4.7.2.2 C-Language Generation Strategy

The C99[97] code generation strategy is to map itineraries to C functions, registers in itineraries are mapped to function-local variables of the correct type and object references are mapped to the architecture's native pointer type. For each class used in the system a `struct` is defined according to the corresponding concrete binary object layout. Each class that has static fields is also generated a 'class instance' `struct` too, and for every class and array type used the integer type identifier defined in a header file.

The itinerary action structure maps very easily onto C. Each action that is an exception handler or could be the target of a branch is assigned a C label and branch actions generate optionally guarded **`goto`** statements.

The C generator is complicated by Java behaviours that change depending on the runtime type of an object. To conserve runtime data memory, which is expected to be the most limited type of memory, the code generator does not use dispatch tables (*vtables*) to lookup which method to execute for a virtual dispatch, and the 'class instance' objects that contain the static fields of a class do not contain any information about their interface implementations or superclass. This means that objects do not need to contain a reference to their specific class's dispatch table.

Each time a type-dependent action is encountered in an itinerary, all of the possible outcomes as determined by the previous application building procedure are enumerated. For virtual method invocations where it is determined the object on which the method is to be executed can only be of one type, the virtual dispatch is eliminated and replaced with a static dispatch instead. Where it cannot be statically determined which dynamic type an object is at compile time, a call to a dynamic dispatcher function is emitted. Chi emits a dispatcher function for every virtually dispatched method invocation with a distinct static reference type; every invocation of a particular method with the same static reference type will share a dispatcher. For example, the `toString()` method that is declared in `Object` can be invoked on all runtime object types, and via any static reference type. This means there can be a large number of dispatchers for `toString()`, and that the more specific the static type of the invocation, the fewer possibilities the

used dispatcher must consider. Internally, dispatchers are large case statements and rely on the optimisation of downstream compilers to chose the best runtime implementation. Only dispatchers that can actually be invoked are generated.

This strategy for virtual method dispatch is only possible as Chi has complete application knowledge, and applications have static type structures as classes cannot be loaded at runtime. This enables interface invocations to be considered the same as any other virtual dispatch. Additional memory is saved as dispatch tables are not required for each interface type. The disadvantage of this scheme is that virtual method invocation *may*[29] be a more complex procedure and requires more code memory than the conventional virtual dispatch mechanisms. When the basic two machine example in figure 4.3 is compiled for a 32-bit intel architecture target, there are 65 used dispatchers cumulatively using 12,687 bytes of code[30]. In contrast, a more complex application (a dining philosophers implementation with five defined philosophers and forks, discussed in section 5.3.4.1) requires a very similar 15,014 bytes for 68 dispatchers, accounting for 4.0% of the applications total compiled code size.

All code generated depends on `stdint.h` and `math.h` for their definitions of the standard integer types and their most extreme values. The model for little endian, 32-bit, POSIX[90] architectures also requires a suite of additional system-defined headers for console access, determining the time, POSIX threads, UDP networking and memory allocation. The models for Xilinx MicroBlaze targets, used for evaluation in the next chapter, have no extra C dependencies as they uses drivers written in Java (hidden to the user application) to provide basic console IO and timing.

The generated C application is entirely self-contained, requiring no 'libchi.a' or similar to be linked against. Each class in the original Java application (and one for each system library class used) is allocated its own C file containing method implementations. All methods, classes and types are declared across three main header files.

4.7.2.3 Minimising Static Memory Consumption

There are two main sources of static memory consumption: code and literal data. Both of these are immutable at runtime so do not need to exist in writable RAM. This also

[29]If the C compiler is particularly efficient at optimising case statements then the dispatcher mechanism could be at least as fast as an ordinary dispatch-via-vtable invocation.

[30]The size of the `.text` section for the `dispatchers.o` compiled object.

means that these components can be safely stored in shared memory.

Code memory consumption is minimised by only emitting functions that can possibly be used. The application exploration procedure is designed to build a sufficient type- and call-graph to understand the complete behaviour of the application under compilation. However, this procedure is conservative as methods are explored that cannot actually be invoked at runtime. The set of required methods can be further pruned by only including methods that meet a 'liveness' test. A method is considered live if:

- It is the Chi runtime's entry point. This is the method that will invoke the application's entry point after system initialisation.
- **OR**
  - If the method is non-static, then the class must be instantiated in the application
  - **AND** The method must be callable by a live method.

Only two types of Java objects can actually exist as literals[31] in a Java class file: Strings and Classes. Class literals are only used by applications for a small fraction of classes, so these are instantiated on-demand by the Chi runtime. String literals are coalesced such that all identical strings in the code will reference the same read-only, non-heap object at runtime.

4.7.2.4 Runtime Operational Components

The main operational components of the Chi runtime are the entry point and the heap allocator. Both are written in Java and undergo translation along with the user's program code. The Chi entry point becomes the generated source code's entry point and performs duties such as initialising the heap allocator, initialising any device drivers that the target architecture uses, calling each class initialiser present in the application, and finally invoking the entry point specified in the user's application. This generated entry point contains the top-level exception handler responsible for catching all `Throwable` objects and printing a message to the system's console (if one exists) if an uncaught exception has propagated out of the user's application.

[31] References to the class's constant pool

Network-Chi has three memory managers that can be selected at compile time: a basic allocator which can never recover memory once allocated, a scoped memory manager that implements a simplification of the Real-Time Specification for Java (RTSJ)[219] scoped memory concept, and a basic garbage collecting memory allocator.

Machine Java's construction (and implied application programming style) requires frequent allocation of transient objects. This excludes the possibility of using a non-reclaiming memory manager. In addition, the lifetimes of objects, such as newly received data items from a network interface, are unknown at compile time and cannot be coerced into a hierarchical usage pattern required by the scoped memory manager. In a scoped memory scheme a machine would only be able to reference a received datum if it was allocated in the same memory scope, or in an enclosing scope, but an interrupt handler receiving an object from another processor could never know which machine the object will eventually be destined. If received data were to be allocated into a machine's enclosing scope, common to all machines, then this would be accessible in an event handler. However, memory exhaustion would arrive quickly as the objects in a memory scope cannot be individually deallocated, and the scope cannot be purged while child scopes still exist. Severe and unacceptable (from a programming complexity perspective) convolutions involving *reusable* objects are required to support a Machine Java type framework with a scoped memory allocator. A more detailed overview of Chi's scoped memory management can be found in appendix E.

4.7.2.5 Garbage Collection

In the context of Machine Java, only the garbage collecting allocator is suitable. Garbage collection facilitates very flexible programming patterns, such as the use of boxed primitives and repeated allocation of event handlers. Garbage collection also enables efficient use of highly limited memory resources. Network-Chi provides a simple non-realtime garbage collector to enable standard Java to be written without fear of memory exhaustion or unsafe references. The garbage collector prioritises memory efficiency over all other concerns, including garbage collection predictability and speed of collection. Real-time garbage collectors have been investigated thoroughly [16, 20, 108, 159, 160] and could be implemented instead if predictability is a more important runtime characteristic.

The important qualities of Network-Chi's garbage collecting memory allocator in-

clude:

- Implementation in Java, allowing the same collector to be used for multiple targets without modification. The collector does assume a 32-bit pointer width, and also requires assistance from the active computational model to determine the stack bounds for all threads and to flush all processor registers to the stack.

- Overall, the collector can be summarised as a non-moving, conservative, stop-the-world, mark sweep collector. This construction favours high maximum memory utilisation (no space is reserved for memory compaction) and avoids memory leaks due to reference cycles (mark-sweep schemes do not use reference counting). On architectures with threads, all threads are paused during a collection but these are not the main focus for Network-Chi. Finally, the collector is *conservative* meaning that it will never collect an object that is referenced but it may fail to collect objects that are not referenced.

- The allocator divides the heap into small blocks and maintains metadata for each block. Each memory block can be used by at most one object, meaning that unless an object requires a perfect multiple of the block size some memory is wasted in the object's last block. The allocator uses one byte of metadata per block with flags for the usage of the block, if the block is the head of an object, if the block is for a `static` instance (the static fields of a particular class), and several bits used by an in progress garbage collection. Using block-based metadata avoids including additional fields within each object instance, at the cost of a static reduction in available heap size. As GC metadata within an object must be at a fixed offset (such as zero), and some platforms require field alignments within objects, a GC metadata field would have to be a multiple of the alignment width. On the considered platforms this would be 32 bits, or $4\times$ the metadata size for a block.

- The collector identifies roots of the reference graph from the static instances identified from block metadata and by scanning the stack for words that appear to be references. This stack scanning procedure is what introduces the collector's conservatism as any value on the stack that has the value of a valid reference by coincidence will keep the referenced object alive. A value on the stack must pass several tests to be considered a reference: It must point to a region within the valid

heap, it must be aligned to point to the start of a block, and the block pointed to must be marked as the head of an object.

- The collector does not defragment memory, but the allocator takes basic precautions to avoid memory fragmentation: single-block objects are always allocated into the lowest index available block (*first-fit* allocation). Larger object allocations use a rolling index to avoid re-scanning the heap metadata for each allocation. This is Knuth's *next-fit* [111] allocation scheme. Although next-fit allocation saves some effort where the object to be allocated is too large to fit into any of the free-regions towards the start of the heap, a number of disadvantages to this scheme have been noted [105, §7.2].

- The collector only supports standard (strong) Java references. Weak, soft and phantom references are not supported. Finalizers, such as Java's `Object.finalize()` method, can only be used safely if they are executed asynchronously [30][32], and this requires that they are executed in another thread. As Chi does not guarantee to provide threads on all platforms, `finalize()` methods are never executed by the Chi runtime.

4.7.2.6 Tuning the Block Size

The memory manager's block size has a significant impact on memory efficiency and runtime performance:

too small If the block size is too small then memory is wasted on excessive metadata and performance will suffer. Smaller block sizes favour memory fragmentation and increase the overheads due to allocation and and garbage collection. This is because the allocator and collector use algorithms that are $O(n)$ where n is the number of allocation blocks.

too large If the block size is too large then memory is wasted when objects do not fill their blocks exactly.

The minimal Chi object on 32-bit architectures is 4 bytes as the hidden class identifier field consumes an integer. Very few classes will have no fields, so the minimum realistic

[32]Boehm provides a good summary of the necessary difficulty of providing object finalizers in [30]

Measurement	nPh	MD5	STHP	ST	Mean
Failed request size (bytes)	136	120	168	152	144
Contiguous free (bytes)	88	80	144	144	114
Used (bytes)	37374	38448	37272	36212	37326.5
Free (bytes)	424	632	888	1808	938
Objects	1988	1515	1857	1752	1778
Mean bytes per block	6.28	6.49	6.33	6.27	6.34
Mean bytes per object	18.80	25.38	20.07	20.67	20.99
Objects <= 8 bytes	460	310	452	391	403.25
Objects > 8 and <= 16 bytes	700	384	606	618	577
Objects > 16 and <=32 bytes	713	551	637	548	612.25
Objects > 32 and <=64 bytes	108	148	137	162	138.75
Objects > 64 bytes	7	122	25	33	46.75
Wastage if 8 byte blocks	3762	2504	3392	3572	3307.5
Wastage if 16 byte blocks	11538	9504	10232	9884	10289.5
Wastage if 32 byte blocks	31874	23952	29704	29068	28649.5
Wastage if 64 byte blocks	91970	67600	84648	79820	81009.5

Table 4.3: *A summary of four applications' heap memory following an* `OutOfMemoryError`*. The wastage figures consider the sum of memory unused in blocks if the application's objects were allocated with the specified block size. The execution platform was a single processor with a 48,000 byte heap.*

size for an object is 8 bytes. This covers objects with a single reference or integer field, or a zero-length array.

To determine the most efficient block size with respect to memory wastage, data on heap usage was collected from a number of Machine Java applications. The purpose of these applications is covered in the next chapter, but for now only the heap usage characteristics are of interest. Each application was executed on a single processor platform with 48,000 bytes of heap memory, and the applications were tuned to eventually run out of memory before completing successfully. When the application had run out of memory the heap was examined to determine the distribution of objects in use by the application. An application is considered to have run out of memory when the garbage collector throws the first `OutOfMemoryError`. This happens when a garbage collection cycle could not free enough contiguous memory to satisfy the most recent request. For the purposes of this experiment an 8-byte block size was used. Table 4.3 summarises the results of this investigation.

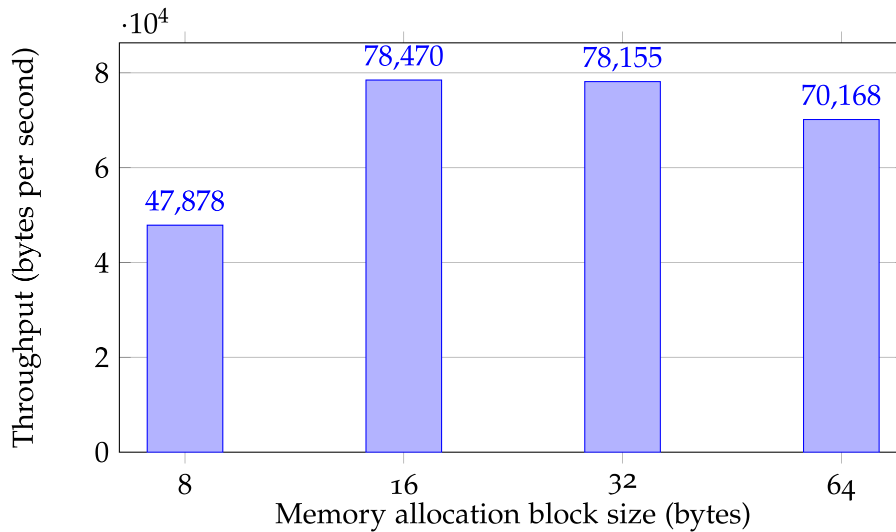


Figure 4.25: *The impact of the memory allocator's block size on overall application throughput. This is the SpeedTest (ST) application with a 5-byte message size, four pairs of communicating machines and executing on a 32 processor platform.*

Table 4.3 provides a breakdown of overall heap usage and free memory at the time of failure, and a coarse indication of the object size distributions. Finally, a calculation of wasted memory due to partially utilised blocks is provided. The wastage figure is the sum of all unused memory in blocks if all of the application's objects were allocated again using blocks of the specified size. The table demonstrates the expected outcome that smaller block sizes reduce partial-block inefficiencies: 8-byte blocks waste an average of 3.3KB[33] but 64-byte blocks would waste 81KB for the same set of objects. To get a complete comparison of memory efficiency the block metadata must also be accounted for. As this is one byte for each block, this adds an overhead of 6KB for 8-byte blocks in a 48KB heap, and 3KB, 1.5KB and 750 bytes of overhead for 16, 32 and 64 byte block sizes respectively. Even with the metadata overhead included the 8-byte block size is most efficient, using 9.3KB overall compared to 13.3KB for a 16-byte block size.

To consider the performance aspects, the SpeedTest (ST) application was tested on each of four potential block sizes. The impact of the allocator's block size on overall application throughput is shown in figure 4.25. It can be seen that there is a significant performance benefit for allocation block sizes greater than 8 bytes. Throughputs for 16 and 32 byte block sizes are within the margin of error and 64 byte blocks appear to suffer a marginal disadvantage.

[33] This document follows the convention that 1KB is 10^3bytes, and 1KiB is 1024^1bytes.

Overall, the substantially better performance achieved when using a 16 byte allocator block size appears to justify the additional memory overhead compared to the 8 byte block size. As Chi's garbage collector is compiled at the same time as application code, only a trivial modification would be required to enable application-directed garbage collection parameters.

4.7.3 Network Runtime

Network-Chi enables standard Java applications to be executed on homogeneous networks-on-chip. Low-level architectural details are handled by Chi's internal 'computational model' mechanism, and this can be selected via a command line switch at compile-time. High-level details, such as the amount of heap memory to use, or the dimensions of the mesh network are also selectable via command line switches. Java applications compiled with Network-Chi have a simple model of execution: Every processor in the mesh (referred to as *nodes*) begins executing the same entry point in the application at NoC startup-time. The entry point can be any class with a
"`public static void main(String[] args)`" method, as in standard Java. Application code is able to query the runtime to determine which node it is executing upon and differentiate its behaviour accordingly. From the application programmer's perspective each node has its own private Java heap with no access to objects on other nodes. Even if the underlying NoC architecture provides shared memory this is not exposed to the application.

The processor at coordinates $(0,0)$ must be the first processor to begin execution, and the Network-Chi runtime will boot and synchronise with other processors if necessary before releasing control to the application's entry point.

A basic 'hello world' example using the framework can be seen in figure 4.26. On startup each node in this example begins executing from the entry point and first queries the framework to determine if it is the *origin* node. All nodes in Network-Chi appear the same to the application and have no intrinsic differences other than their address within the network, the framework only identifies the node at coordinates $(0,0)$ as the origin node for convenience. In the example, the non-origin nodes send `HelloMessage` objects to the origin node, and the origin node will wait until it has received a message from every other node. This example assumes a fully populated rectangular array of nodes, but this is not required in general by the Network-Chi framework.

4.7.3.1 Execution in a JVM and on POSIX Operating Systems

Network-Chi applications can execute in a standard JVM for basic functional verification. Execution in a JVM is not a NoC simulator and may behave differently to a real hardware NoC; no attempt is made to emulate timings, router contention or other NoC behavioural artefacts. The runtime implements the network API's behaviour in a JVM by detecting the first call to any API method and then constructing thread for every node of a notional rectangular NoC. These node-threads begin executing from the same entry point that the JVM was started with, and their coordinates on the 'NoC' are recorded in a table. When executing in this mode a network with large and lossy buffers between nodes is assumed; sends will never block. The communications between nodes is implemented using UDP networking to enable interoperability with the 'POSIX' compilation target that also uses UDP as its communications protocol.

Network-Chi can be used to compile a Java application for execution as a standalone binary on a POSIX operating system. In this case the Network-Chi runtime on the origin node spawns another process for each node in the emulated network. The processes identify their coordinates from a command-line argument provided by the origin node. All subsequent communication between nodes is conducted via the network API, which in turn uses UDP sockets. Compilation to a binary has several advantages over execution in a JVM:

- The memory usage of the application is rigidly enforced. In a JVM memory usage cannot easily be bounded on a per-node basis as all threads in a Java application share the same memory.
- Applications cannot 'cheat' and communicate via shared memory. When executing in a JVM, **`static`** fields are shared between all nodes, but this is not possible when each node executes in an isolated operating system process.
- Application compatibility with Network-Chi can be assured. Network-Chi has complete Java *language* support, but does not support a large proportion of the standard libraries due to dependency explosion and native methods.

4.7.3.2 Network API

The Network API (see table 4.4) is intended to be minimalist while remaining congruent with the nature of NoC hardware. An example of its usage can be found in figure 4.26.

```
1  public static void main(String[] args) {
2    if (amOrigin()) {
3      log("I'm the origin!");
4        int rxCount = 0;
5        while (rxCount<width()*height()-1) {
6            HelloMessage next = (HelloMessage) receive();
7            log(next);
8            rxCount++;
9        }
10       log("All nodes reported in!");
11   } else {
12       send(0,0, new HelloMessage());
13   }
14 }
```

Figure 4.26: *A basic 'HelloWorld' application written using the Network API. The origin node (coordinates 0,0) will print a message it receives from all other nodes and then quit. The definition of the `HelloMessage` class can be seen in figure 4.27.*

Method	Description
`void send(int, int, Flattenable)`	Sends any object that implements the `Flattenable` interface to a specified X and Y coordinate. May block until received by the remote node. May throw a `MessageTooLargeException` if sending the Flattenable will exceed the underlying network's MTU.
`Flattenable receive()`	Receives the next object from the network. Will block until an object is received.
`Flattenable tryReceive()`	Receives an object from the network if one is available, or **`null`** if no object is received.
`int myX()`	The X coordinate of this node.
`int myY()`	The Y coordinate of this node.
`int width()`	The width of the network.
`int height()`	The height of the network.
`boolean amOrigin()`	Returns true if this node is at coordinates $(0,0)$
`void log(Object)`	Writes a specified message to a debug console prepended with a node identification string. This guarantees to not permanently allocate any memory regardless of the Object's `toString()` behaviour.
`void setReceiveHandler(Runnable)`	Sets an asynchronous object arrival handler.
`void setNodeLocalTag(Object)`	Sets guaranteed node-local variable.
`Object getNodeLocalTag()`	Retrieves the previously set node-local variable.

Table 4.4: *The API provided to applications for communication via a mesh network. These are all static methods of the `Network` class.*

In this example the Network API has been statically imported to make the code clearer to read.

Nodes can send messages to other nodes via the `send(int, int, Flattenable)` method which accepts the X and Y coordinates[34] of the destination node and a reference to *any* Java object that implements the `Flattenable` (see section 4.7.3.4) interface. This is the only provided mechanism for communications between nodes and is similar to the *message* communication abstraction provided by MCAPI[202]; messages are sent as fixed size datagrams and communication appears to be an atomic operation to the application.

When a destination node invokes `receive()` or `tryReceive()` a copy of the sent object will be returned. The API does not provide a mechanism to determine which node sent the message, but the sender is able to include their own address in their message if they choose. To better reflect the nature of many underlying network architectures the API is deliberately asymmetric. There is a non-blocking message read method but there is not a non-blocking send method. This is because many network router designs (including the Blueshell example hardware used in the next chapter, section 5.2) do not provide any mechanism to determine if a remote node is ready to accept a message without first sending a message, which obviously may not be accepted.

While the API declares that `send()` may block the caller until the receiving node is ready to receive a message it is also not guaranteed to do so. If there is enough buffer for the object available on the sending node or in the network routers between the sender and receiver, the implementation of the API is free to return the `send()` call as soon as it can. The API also does not require that messages arrive in the order that they were sent if they were sent from different nodes; the API does not establish a NoC-wide global notion of time or ordering of messages, but it does require that messages from a single node to another node arrive in order.

No methods of the networking API will permanently allocate memory, with the exception of the `receive()` and `tryReceive()` methods which will allocate the objects which they return.

```
1  public static class HelloMessage
2          implements Flattenable {
3    private final String msg;
4    public HelloMessage() {
5    this.msg = "Node "+myX()
6         +","+myY()
7         +" reporting in!";
8    }
9    public void flatten(DataOutputStream out)
10   throws IOException {
11   Utilities.flatten(msg, out);
12   }
13   public HelloMessage(DataInputStream in)
14   throws IOException {
15   msg = (String) Utilities.inflate(in);
16   }
17   public String toString() {
18   return msg;
19   }
20 }
```

Figure 4.27: *The 'HelloMessage' class referenced in figure 4.26, demonstrating an implementation of the `Flattenable` interface.*

4.7.3.3 Object Serialisation

Effective object serialisation between processors is critical for the implementation of high-level message passing frameworks like Network-Chi. In an embedded context, such as a processor on an FPGA implemented NoC, the structured property of objects and the desire to perform deep (recursive) serialisation on an object causes several significant issues:

- Expensive metadata is required at runtime to identify the fields in a class that refer to other objects, so that these referenced objects can also be serialised. To reconstruct an object, special serialisation constructors must be provided that allocate memory but do not invoke any declared constructor in the class. Both metadata and serialisation constructors are usually provided by Java reflection which is not a justifiable overhead in embedded contexts. To an extent this issue can be addressed by having the compiler automatically generate serialisation methods for

[34] The framework uses a 2-dimensional (x,y) addressing scheme to reflect common NoC topologies, but can be easily extended to support any absolute addressing scheme, such as (x,y,z) or (r, θ).

all possible serialisable classes, however this moves the burden from metadata to increased code volume.

- With or without metadata, object structure dictates that hardware offload such as DMA cannot be used for deep serialisation as objects are not guaranteed to be contiguous in memory. Complicated schemes can be contrived to perform a DMA operation for each shallow object, but still requires CPU intervention for every object in the structure. Direct memory copying of objects is also only appropriate between processors with identical in-memory representation of values, which may not be true in heterogeneous NoCs.

- Objects are unlikely to be reconstructed into the same memory addresses that they were located at on the source node, so upon reconstruction all references in each object must be updated to reflect the new addresses of the received objects. As with serialisation this only provides yet more work for the CPU to perform that cannot be offloaded to a DMA unit, further increasing the overhead.

In short, object transfer between nodes is complicated because of a lack of similarities; neither data representation nor memory location can be assured between nodes in general. If more assurances of similarity can be made, serialisation can become more tractable.

- **If representation differs** objects must be fully serialised by converting all fields into an exchange format. This is the approach both standard Java and Network-Chi use. It is extremely expensive as it requires CPU attention for all data.

- **If only location differs** objects can be individually shallow copied amongst nodes, but references require fixing.

- **If neither representation nor location differ** objects can be individually shallow copied amongst nodes.

- **If assurances of contiguous allocation can also be made** objects can be deeply copied amongst nodes in a single transaction, this would enable object-based message passing as cheaply as communication of unstructured buffers.

In the context of a homogeneous NoC it is possible to assume identical representation between nodes, but a scheme to ensure identical references between nodes remains future work.

4.7.3.4 Flattenable Objects

The `Flattenable` interface is the compromise made by Network-Chi between full support for automatic object serialisation and no support at all. This approach avoids the need to retain expensive class metadata at runtime, at the expensive of programmer effort. A class that implements `Flattenable` must provide a method `flatten( DataOuputStream out)` to write the fields it cares about to the provided stream, and it must also provide a *reconstruction constructor* with a single `DataInputStream` argument to construct a new instance with information retrieved from the specified stream. The Java compiler cannot check for the existence of the reconstruction constructor as Java interfaces cannot specify constructors, but Network-Chi will enforce this contract during the compilation workflow. An example of a simple `Flattenable` object is given in figure 4.27.

A utilities library is provided to assist programming with `Flattenable` objects:

- `void flatten(Object, DataOutputStream)` will emit the object supplied into the output stream if possible. Only `null`, `String`, and `Flattenable` objects can be flattened. The implementation emits a tag into the stream before the object to enable identification of the stream contents, and a class identifier is also emitted if it will be a `Flattenable` object next.

- `Object inflate(DataInputStream in)` reads from the input stream and reconstructs the `null`, `String`, or `Flattenable` that is next in the stream. If the implementation reads a `Flattenable` tag from the stream, a compiler generated psuedo-constructor (`inflateFlattenable`) is invoked. Chi automatically generates a body for `inflateFlattenable` that can invoke the reconstruction constructor of any object implementing `Flattenable` used by the application, determined by the class identifier received. The provision of this pseudo-constructor avoids forcing the programmer to write a reconstruction factory manually, ensuring that if an object implements `Flattenable` the programmer can send it via the networking API with no additional effort.

- For convenience, `Flattenable` wrapper classes are provided for all primitive arrays, collections, sets, lists, and maps. The data items contained within the `Flattenable` collections must also be either `null`s, `String`s, or `Flattenable`.

The `flatten` method and reconstruction constructor of the `Flattenable` interface could be automatically generated by the compiler in the future.

4.8 Summary

This chapter introduced a new *machine-oriented* programming framework: *Machine Java*. Machine Java realises all three of the Machine Abstract Architecture models discussed in previously in chapter 3. Building upon the Java programming language, Machine Java enables the description of complex applications that can be compiled for execution on resource constrained MPNoC architectures. The use of a high-level general purpose language enables a great variety of existing tools and methodologies to be re-used in a new architectural domain.

The Java programming language already embeds a threaded model of concurrency and contains an assortment of features for synchronising threads and guarding access to shared objects. All of these features are redundant for applications designed in Machine Java as the framework guarantees that machines can never share the same objects. Machine Java additionally guarantees the sequential consistency of code executed within a machine, simplifying the task of designing reliable machine code.

The exploitation of Java's generic type system enables the compile-time validation of inter-machine type-safety and also ensures that machines have consistent expectations of the communications protocol.

Ordinary Machine Java applications must adhere to a set rules concerning the accessibility of machine class members but the isolation of machines is retained even when the rules are accidentally violated or deliberately disabled. Machine Java verifies compliance of application code with the rules as an *assistance* to programming rather than a hinderance; the behaviour of non-compliant code likely to be platform specific.

Machine Java enables dynamic application architectures to be defined that can alter their runtime structure in response to changes in application requirements. Applications can request new machines freely but the circumstances in which machines can be destroyed are far more limited, and just as Java objects cannot be explicitly deallocated, neither can machine instances. Without careful design applications that allocate machines dynamically may unintentionally exhaust the platform's available resources.

The runtime library overhead required to enable dynamic applications is significant but is addressed in part by the presented *Network-Chi* compilation strategy. This enables the execution of Machine Java applications on processors with very little memory by only emitting code that can certainly be used by the application, and a by using a minimal in-memory representation of Java objects.

Chapter 5

Evaluation

Contents

4.1 Java . . . 129
4.2 Machine Java . . . 130
4.3 Machine Oriented Programming . . . 131
4.3.1 Single-Thread Equivalence . . . 134
4.3.2 Machine Classes . . . 135
4.3.3 Machine Class Restrictions . . . 140
4.3.4 Machine Life Cycle . . . 144
4.3.5 Machine End-of-Life . . . 147
4.4 Platform Agnostic Framework . . . 150
4.4.1 Communications Channels . . . 150
4.4.2 Implementing Channel Protocols . . . 160
4.4.3 Spontaneous Event Sources . . . 168
4.4.4 Processor Managers . . . 169
4.5 Platform API . . . 174
4.5.1 Platform Classes . . . 176
4.5.2 Processors . . . 178
4.5.3 Resources . . . 179
4.5.4 Communications . . . 179
4.5.5 The XYNetwork Platform . . . 180
4.6 Implementation in Standard Java . . . 184
4.6.1 Starting the Machine Java Framework . . . 184
4.6.2 Machine Instance Identification . . . 186
4.6.3 Machine Creation and Channel Addressing . . . 186
4.6.4 Machine Isolation and Communication . . . 189
4.7 Implementation on Bare-Metal Hardware . . . 190
4.7.1 Compilation Strategy . . . 193
4.7.2 Language Runtime . . . 204
4.7.3 Network Runtime . . . 213
4.8 Summary . . . 221

5.1 Overview

Chapter 3 of this thesis presented the Machine Abstract Architecture (MAA), a set of models for the structure of *machine oriented* applications, a simplified structure of the execution platform and the limited connection between the two provided by a framework. Machine oriented applications are platform-independent but still allow varied communications channels to be defined between their constituent machines. Machines are defined by their temporal and spatial isolation from one another, and this forms the basis of arguments for scalability: Machines are only affected by their explicit interactions with other machines, and these interactions are limited to the local areas of the interacting machines. All other machines in the application and processors in the platform have no impact on a machine, and therefore it does not matter how many of these entities there are, or what types they happen to be. Similarities between the structure of the platform and application models encourages the interpretation of an application as an idealised *virtual platform* for the behaviour that it implements.

In chapter 4, the Machine Java programming framework was introduced. Machine Java takes advantage of the processing and memory abstractions of the Java programming language but replaces Java's concurrency abstractions with the machine oriented abstraction of the MAA's application model. Machine Java's construction enables Java's strong and static type system to be applied across a distributed application without the need for Java code generation or additional verification tools. Java's type system is also used to encode the protocol characteristics of communications channels, avoiding Internally, Machine Java has a layered architecture that not only separates the application from the platform, but also separates the machine oriented API from the hardware specific realisation. The limited resources available in resource constrained MPNoCs, and the restrictions imposed by existing 'small' Java implementations motivated a new strategy for Java execution. In section 4.7 an optimising, whole-application Java compiler was presented to support the implementation of dynamic programming frameworks, such as Machine Java, on highly limited processing platforms. *Network-Chi* provides comprehensive Java language support while also substantially minimising the static and dynamic memory consumption of its output. Machine Java's more unique requirements are addressed via explicit support for non-shared memory multiprocessor networks and inclusion of limited reflection functionality.

A primary claim of the machine abstract architecture is that applications are platform-

independent if the realisation of the MAA is also platform-independent. Machine Java, as a realisation of the MAA, supports this claim through two arguments:

- A Machine Java application will execute correctly without modification on any platform supported by Machine Java.
- Machine Java is capable of execution on multiple architectures, and therefore will support multiple platforms:
    1. Machine Java applications are not just *descriptions* of an application that conform to the application model, but they are fully executable Java applications and the Machine Java framework is also an executable implementation. As Java itself is capable of executing on a vast array of different architectures, it follows that an application would be capable of execution on a wide variety if different architectures.

       As with every other target architecture, execution in a JVM requires a platform description that can provide sensible internal API drivers for a JVM. The most basic universal platform would describe a single processor with unlimited resources. This argument is not particularly satisfying as the intended domain for this thesis —resource constrained non-cache-coherent MPNoCs— are incapable of hosting a recognisable standard JVM. Standard Java implementations are too large and depend too strongly on coherent memory between threads.
    2. A platform implementation has been described which can allow Machine Java to execute using the Network-Chi's abstraction for homogeneous mesh networks. This platform is also executable in a standard JVM using separate Java threads to emulate each processor of the mesh, but it further provides the ability to target platforms which provide a POSIX operating system interface. This compilation stage enables memory constraints to be enforced, somewhat validating the claim of applicability to resource constrained networks. The assumption of a POSIX OS is also not well suited to highly restricted target architectures.

Using Network-Chi to execute a Machine Java application on a commodity OS, such as linux, confirms only basic functional correctness and resource constraints. This is problematic as only the memory used by the compiled application is constrained by

the compilation process; the memory usage of the runtime and OS are not entirely accounted for by this procedure. Network-Chi does not attempt to simulate the timing or contention aspects of a mesh network, so there is also the possibility that the emergent behaviour of a real mesh network would result in a different functional outcome from an application. This is especially likely if an application relies on specific inter-message ordering between multiple machines, but this could be considered a fault in the application's design. The POSIX OS target is also unable to expose architecture specific failure modes such as deadlock of a mesh network. Hardware implementations that are susceptible to deadlock may not be able to accommodate arbitrary Machine Java applications as a degree of platform awareness may be required to avoid deadlocks. Therefore, to provide a realistic evaluation Machine Java is also applied to a true resource constrained network-on-chip architecture: *Blueshell*.

In the remainder of this chapter aspects of machine modelled programming and Machine Java are evaluated against the overall goal of supporting application development on resource constrained multiprocessor networks. The hardware platform used to validate the implementation strategy is described in section 5.2. In section 5.3 the construction of applications within Machine Java is considered, with particular attention to the additional challenges introduced. In section 5.4 the performance and implementation overheads are considered.

5.2 Hardware Evaluation Platform

In this section the *Blueshell* hardware evaluation platform is described. As Blueshell is actually a network generator rather than a specific instance, the general characteristics of Blueshell networks are described first. The specific network instance used for evaluation is detailed in the next section (5.2.2).

5.2.1 The Blueshell NoC Generator

> ***Note:*** *The Blueshell NoC generator described in this section was primarily the work of Jack Whitham with contributions from other members of the Real-Time Systems group. The Blueshell NoC generator is not claimed as my work, nor as a contribution of this thesis. It is documented here to provide context and aid experimental reproducibility.*

Blueshell [165] is a collection of libraries written in Bluespec System Verilog [28] to generate multiprocessor networks for implementation on FPGAs. NoCs constructed

with Blueshell are composed of:

- A set of *tiles* that provide some functionality to the network, including Xilinx MicroBlaze [228] processors, memories, ethernet controllers and debug tiles.
- A *Bluetiles* mesh network to interconnect the tiles.
- A *Bluetree* memory network to multiplex access to external memory between tiles requiring access.

The Bluetiles communication network is a very simple Manhattan grid network constructed from Bluetiles routers that are addressed by their X and Y coordinates. Each Bluetiles router has five 32-bit bidirectional physical ports, corresponding to north, south, east, west and a 'home' port which connects a local tile to the router. Each Bluetiles network packet starts with a header declaring its destination address, service port number (see below) and payload length in flits. The packet is wormhole-routed[1] in X then Y order through each router towards the designation tile at a rate of one flit per clock cycle. Bluetiles flits are always 4 bytes transmitted as a word. A packet in transit retains exclusive control over both the input and output ports of the routers that it is travelling through. Other packets cannot flow through a router and are blocked if they would require a port that is already in use; Bluetiles routers do not implement any form of flow control. A packet does not leave the Bluetiles network until it has been fully read-in by the tile connected to the destination router's home port. In many respects the simplicity of Bluetiles makes it a model NoC for software development, as many of the issues for NoC programming (such as network deadlock) are a very real concern.

All Bluetiles packets contain a 'service port' field, analogous to the port numbers used by the internet protocols, to identify the purpose of the message. The port number is used by a recipient tile to interpret the incoming message. Well behaved tiles implement as many of the common Bluetiles services (seen in table 5.1) as appropriate. The mechanism a tile uses to implement a network service does not matter; some tiles implement their services in hardware whereas processor tiles are likely to use software implementations.

[1] *Wormhole routing* or *wormhole switching* is where network packets are divided into smaller *flits* (FLow control unITS). Wormhole routed networks apply flow control and buffering at the flit-level rather than for whole packets. See [80, §F.5]

Service	Port	Description
blackhole	0	Discards all data that arrives.
read	1	Reads memory from the tile and replies with the value.
write	2	Writes values into the tile's memory.
ping	3	Sends the packet received back to the sender.
boot	4	Instructs the tile to boot (branch) to the vector specified.
interrupt	5	Instructs the tile to forward its interrupts to the sender via the network.
message	6	Writes the packet to a debugging device if present
terminate	7	Indicates that the networked application has terminated. A host process communicating with the network can finish and the FPGA can be deprogrammed.
object	20	Indicates the payload is a flattened Java object implementing the `Flattenable` interface.

Table 5.1: *Common services implemented by tiles on the Bluetiles network. Peripheral tiles can be expected to implement **read**, **write** and **interrupt**, whereas a processor hosting the Machine Java runtime will implement all except **interrupt**, **message**, and **terminate**.*

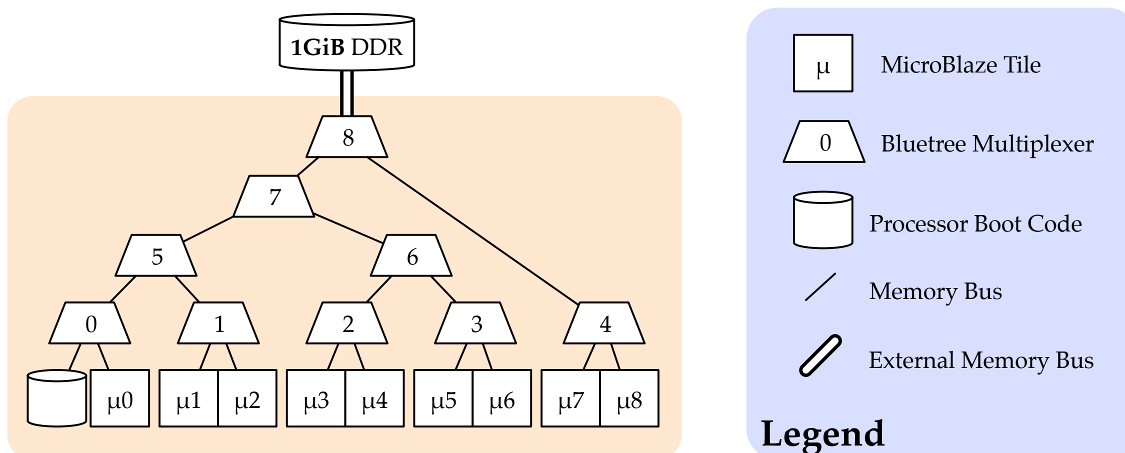


Figure 5.1: *The layout of the Bluetree memory network for a 3x3 network of MicroBlaze processors.*

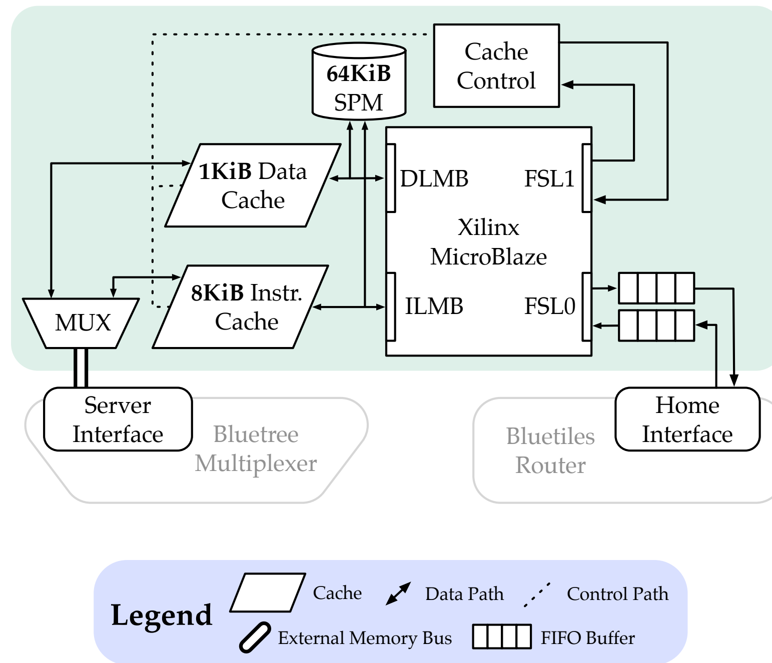


Figure 5.2: *The internal layout of a Xilinx MicroBlaze processor tile. The configured sizes of evaluation platform's memories are indicated.*

A Blueshell instance also contains a second network (*Bluetree*) exclusively for access to shared off-chip memory (such as DDR SDRAM). The memory network reduces contention across Bluetiles and increases the overall flexibility of the platform.

This network is constructed from a tree of 2:1 full duplex *Bluetree* multiplexers where the leaves of the tree are the devices that need access to shared memory. In the case of a 3x3 mesh of MicroBlaze tiles, the memory tree is configured as shown in figure 5.1, and figure 5.2 shows how the Bluetree network is connected to a MicroBlaze tile internally.

While the Bluetree multiplexers hide the topology of memory accesses from clients they do nothing to ease memory contention and do not provide any synchronisation guarantees. Applications must always be mindful of the performance and concurrency implications of accesses to external memory.

General purpose computation is supported on Blueshell networks by Xilinx MicroBlaze processors encapsulated into a tile with caches, local memory and basic peripherals. Each microblaze tile is essentially a complete system on chip. The internal structure of a microblaze tile can be seen in figure 5.2. Each MicroBlaze processor is connected to the Bluetiles router via a Fast Simplex Link [226] (FSL) port. The processor accesses memory through specially constructed instruction and data caches that can be invalidated on a line-by-line basis by the 'Cache Control' peripheral attached to the second FSL.

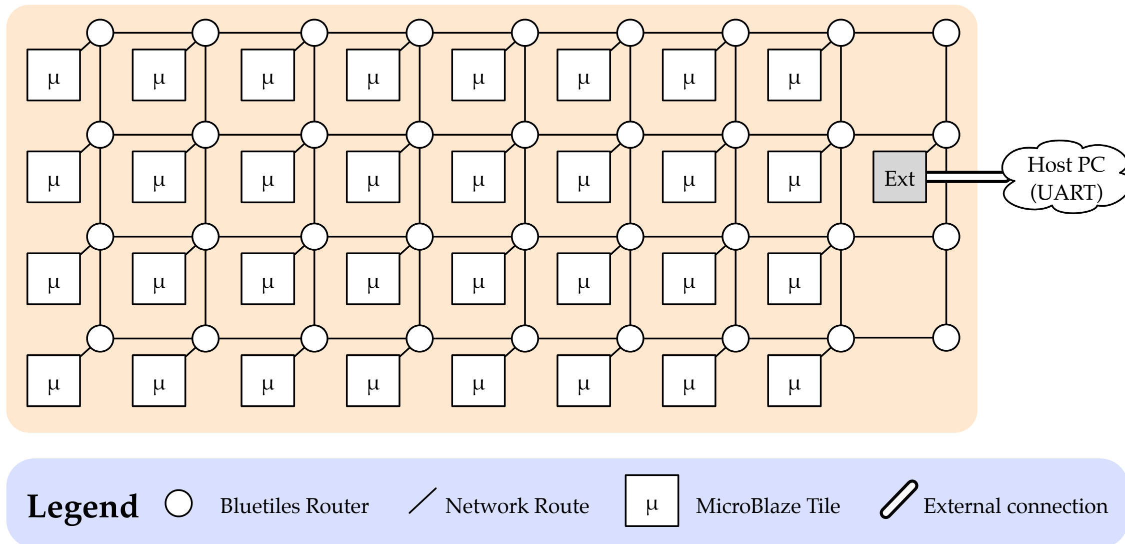


Figure 5.3: *The communications network and processors in the 32-processor evaluation network-on-chip. The communications interconnect has a clock frequency of 100MHz and 32-bit channel width, yielding a bisection bandwidth of 1.6GB/s (1.49GiB/s)*

The cache control unit also acts as the MicroBlaze's interrupt controller and provides a 64-bit clock-cycle counter facility. Each microblaze processor has exclusive access to a fast and local scratchpad memory (SPM). Although the scratchpad memory provides single-cycle access to data and instructions, it has a very limited capacity as it must draw from the FPGA's highly contended block ram resources.

5.2.2 Evaluation Platform Specification

The specific blueshell instance used to validate Machine Java can be seen in figure 5.3 and has the following characteristics:

- An 9×4 mesh network of bluetiles routers, populated with 32 Xilinx MicroBlaze processors in an 8×4 grid. The network is connected to a host PC via a standard UART for communications and bootloading.
- The communications network is clocked at 100MHz, has 32-bit wide connections, and transfers one 32-bit word on each clock cycle. This results in a point-to-point bandwidth of 400MB/sec and a network bisection bandwidth of 1.6GB/sec (1.49GiB/s), if the network were bisected into two 4×4 processor networks.
- The MicroBlaze processors are configured to be fully featured microcontrollers without hardware supported memory protection. In addition to the high-performance

5-stage pipeline, the processors are configured to include: extended floating point units, extended (64-bit) multipliers, hardware dividers, hardware exception support, barrel shift instructions and large branch target caches.

- The MicroBlazes are version 8.50.b and are also clocked at 100MHz. The back-end C compiler used to target these processors was Xilinx's `mb-gcc` version 4.6.2.[2] No peripheral support libraries or operating systems were used; only Xilinx's C runtime (crt0) is used.

- Each MicroBlaze benefits from 64KiB of SPM, 1KiB of data cache for shared memory access, and 8KiB of instruction cache for shared code. The shared memory network provides each processor access to 1GB of DDR RAM. This shared memory is writable by all processors but is treated as a read-only memory for code during application execution. String literals are the only Java objects that will exist in external memory at runtime.

- Each processor uses 16KiB of SPM for Java's call stack and the remaining 48KiB for the local Java heap. As Network-Chi's garbage collector is configured to use a 16-byte block size this implies that there can be at most 3072 Java objects per processor.

- The network is hosted on Xilinx Virtex-7 XC7VX485T [232] FPGA mounted on a VC707 evaluation board [231]. This FPGA contains 486,000 logic cells across 76,000 slices. The evaluation network uses a significant fraction of the FPGA's resources:

  Block Rams 994 of 1030 (96%)

  LUTs 276K of 304K (90%)

  Flip Flops 237K of 607K (39%)

It can be seen that the size and capabilities of the network are primarily constrained by the FPGA's block ram resources which directly limits the local SPM and cache capacities for each processor. Larger or smaller instances of Blueshell networks can be constructed according to the size of the target FPGA.

[2] 20111018 (Xilinx 14.4 Build EDK_P.41 8 Oct 2012)

5.3 Programming with Machine Java

This section explores some of the challenges of designing applications with Machine Java, and by extension the general challenges implied by machine oriented programming. There are few unambiguously positive consequences to programming with Machine Java; most of Machine Java's differentiating characteristics have an equivocal impact on the design of an application and the effort required to do so.

If the assumptions on which Java is built are accepted, then Machine Java's distributed type safety and the ability to run on a Java virtual machine can also be accepted without further consideration. Other than the overheads required to use the Java programming language, there are no readily apparent disadvantages to the static enforcement of types across a distributed Machine Java application. Likewise, the ability to execute on any compliant JVM cannot be seen as a disadvantage but is a somewhat marginal benefit.

A number of positive claims can be asserted about Machine Java and these claims lead readily to a set of evaluation criteria. The evaluation criteria for Machine Java include:

- To what extent are applications actually platform independent? (discussed in section 5.3.2)
- To what extent can application structure be extracted? (discussed in section 5.3.1.1)
- What are the implications of programming without shared memory related synchronisation mechanisms? (discussed in section 5.3.4)
- To what extent are applications fault tolerant? (discussed in section 5.3.5)
- To what extent does event driven code reduce overheads? (discussed in section 5.3.1.4)
- Is machine oriented programming appropriate for time-driven behaviour? (discussed in section 5.3.3.2)
- To what extent are varied application structures expressible? (discussed in section 5.3.3)
- Is it beneficial for application structure to be defined by the code rather than an external description? (discussed in section 5.3.1)

5.3.1 Defining, Extracting and Fixing Application Structure

Machine Java applications do not use an external structural definition and so the structure of an application is entirely defined by its Java code. This distinguishes Machine Java from some other approaches to the description of distributed applications, such as Ptolemy II [3, 118] or AUTOSAR system descriptions [18, 29], as these require the specification of the application's structure in XML with references to the executable code in Java and C respectively.

Machine Java allows an application's structure to be explicitly described by the programmer: machines are clearly defined as subclasses of `Machine` and their interactions are captured by the use of channel classes which must implement the `Intermachine` interface. This type-centric encoding of an application's structure has the benefit that only one language is needed to describe an application, and this avoids the possibility of inconsistency between an applications structural and behavioural descriptions.

However, the combined structural and behaviour description that Machine Java facilitates may not be as clear or as expressive as separated descriptions. In particular the application model only specifies the interconnection of machine *types* and not the runtime layout of an application's machine instances. This allows an application some freedom to make choices about its runtime layout in response to operating conditions, such as available resources or the type of workload. The most significant disadvantage of combining an application's structure into its Java code is that the structure is not readily accessible; it has to be extracted from the code.

5.3.1.1 Extracting Program Structure

In order for a Machine Java application to be executable, its program code must describe some machines. The machines must contain some channels and some number of event handlers to implement the required behaviour. As the machine types are statically defined, and channel instances can only be accessed via the final fields that define them. This means that it is possible to extract a graph of dependencies between machine types without knowing the machines that will exist or their runtime interconnections. As machine types are represented by Java classes, and their instances by objects, there is a clear similarity between the extractable static structure of a standard Java application, and the extractable static structure of a Machine Java application.

The extracted structure of an application has a number of potential uses, for example:

- To determine the viable subsets of machines that could be allocated to a processor.
- To determine the worst case resource consumption.
- To enable static allocation of machine instances to processors ahead of time.
- To statically verify an application's ability to execute on a specific platform.
- To generate a static communications schedule.

5.3.1.2 Determining A Processor's Viable Machine Subset

It is not necessary for every processor in a platform to have access to the code for every machine in an application. Given an application's static machine dependency graph (see section 3.2) and the machine types that the processor is expected to host, the necessary machines' code required can be calculated. Assuming execution strategy where machines execute their code as written, such as in a JVM, or compilation via Network-Chi, a processor must have access to code for every machine type referenced by an instantiated machine instance:

$$ms_p = \bigcup_{m \in \mathbb{M}_p} \text{dependents}(m)$$

where:

- ms_p is the set of machine types for which processor p must have code.
- $\mathbb{M}_p$ is the set of machine types that processor p will host (instantiate) one or more instances of.
- $\text{dependents}(m)$ is a function that provides the set of machine types that machine type m depends on. $\text{dependents}(m)$ is the set of machines directly reachable from m in the machine dependency graph.

An example drawing upon the water tank level control application from chapter 3 is shown in figure 5.4. In each of the configurations A-E a different machine type is selected to be *instantiable* on some target processor(s). Configuration F illustrates the simplest case where the entire application is available to the target processors. Table 5.2 provides the code and data sizes after compilation for the Blueshell target and with non-essential Java features disabled. The Machine Java code for the simplified water

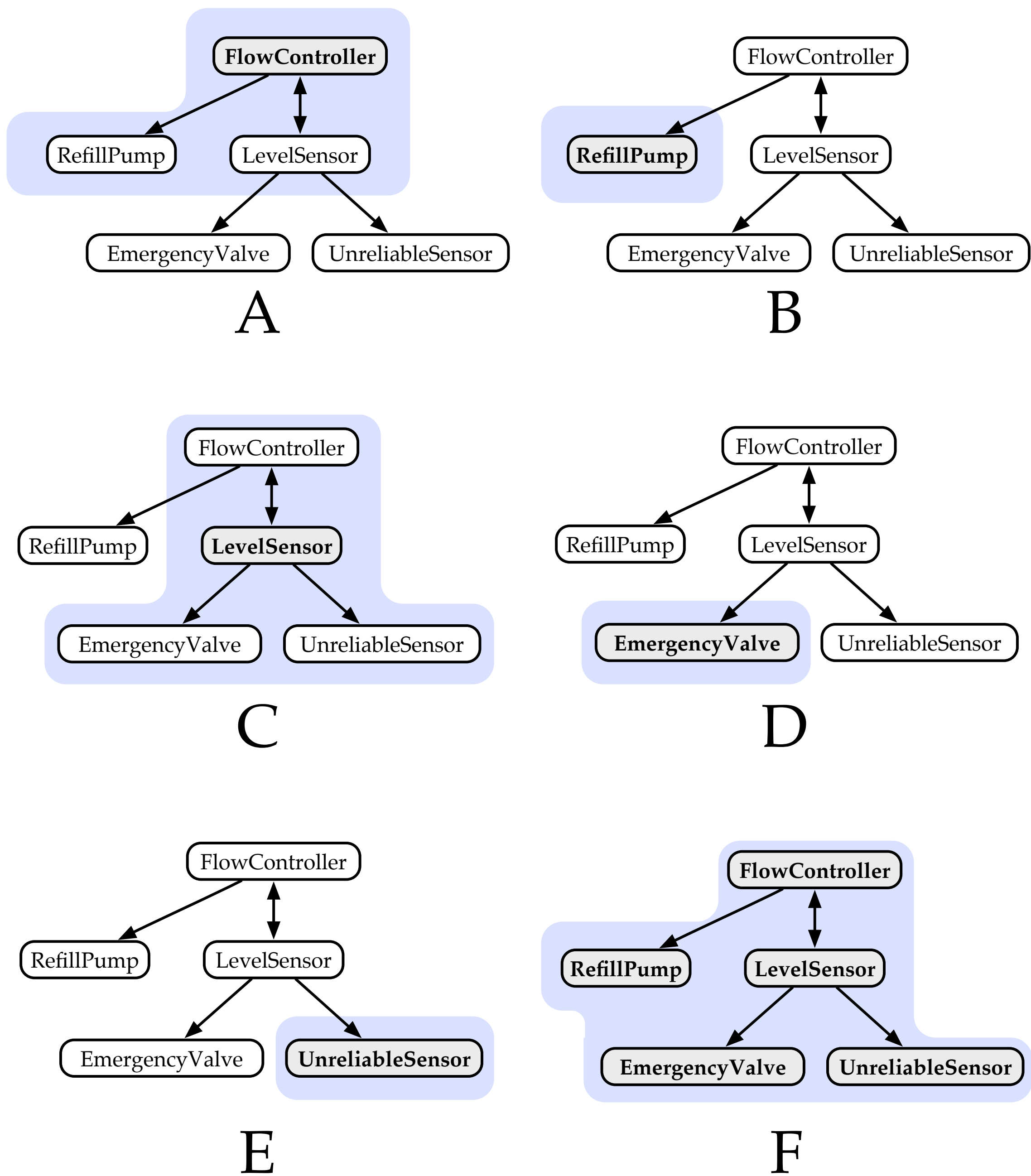


Figure 5.4: *The shaded regions indicate the machines' code necessary to host the machine(s) shown in bold. This is based on the example application in figure 3.2 in section 3.2. This machine dependency graph was extracted automatically from the Java application. The binary sizes required for each of these regions are shown in table 5.2.*

Configuration	**Hostable Machine(s)**	Java classes[3]	(bytes) .text	.data	total	Δ
–	*none*	191	147,000	11,500	158,500	0
A	**FlowController**	202	159,600	11,800	171,400	12,900
B	**RefillPump**	193	148,100	11,700	159,700	1,200
C	**LevelSensor**	207	161,000	11,900	172,900	14,400
D	**EmergencyValve**	195	147,800	11,600	159,400	900
E	**UnreliableSensor**	194	148,800	11,600	160,400	1,900
F	*all*	209	162,000	12,000	174,000	15,500

Table 5.2: *The binary sizes required for each of the configurations shown in figure 5.4 are rounded to the nearest 100. The simplicity of this example application is overwhelmed by the complexity of the Machine Java and Network-Chi runtimes.*

tank application is reproduced fully in appendix B. This application is only intended to be a structural illustration; it does not contain a realistic control algorithm nor code to interact with real sensors.

It can be seen in table 5.2 that the code size overheads due to the Machine Java and Network-Chi runtimes dominate the code and data requirements of the application. However, by considering the difference between each of configuration A-F's total sizes and the size for the application without any machines, it is clear that restricting the machines that are instantiable by a processor can save some amount of code and data space. This example application contains little more than communications channels, and the implementations of the runtime support for these channels requires the nearly the same code burden regardless of the application complexity. It is reasonable to expect that more realistic applications would account for a greater fraction of the output binary size.

Extracting an Application's Machine Dependency Graph

A machine dependency graph represents the statically determinable interactions between machine types. A machine type A is said to interact with another type, B, if it:

1. Requests a new instance of B via any method.

[3]This figure includes all anonymous, system, library and runtime classes.

2. Initiates an interaction with an instance of B via a channel defined in B.

Communication via the return path of a bidirectional channel does not introduce a machine type dependency between the server (the machine replying) and the client (the originator of the query). This is because the necessary details of the return path (the behaviour, type and destination) are specified by the channel and not the return path recipient machine type.

The valid construction of machines (see figure 4.6) enable certainty that only an execution context associated with a machine instance of type A can ever execute code defined in A. This is in contrast to standard Java classes where there are no static guarantees about which thread of control can execute its methods. The coupled code and execution context enable the assertion that if a machine type's code is statically capable of interacting with another machine, then that static capability only applies to instances of that machine.

The static interactions of a machine class cannot be determined reliably by simple inspection of the classes Java source text; a finite regular expression cannot be constructed that could correctly extract all possible channel accesses from the text. However this can be achieved by parsing the Java source code but this is unnecessary as the Java compiler already does this work. Only Java bytecode needs to be examined to build a pessimistic machine dependency graph.

Pessimism is introduced when an application uses shared code that interact with machines or contains event handlers. These *machine aware* libraries would have no clear relationship to any single machine type. In the case of libraries that interact with machines but do not use event handlers, the application's method-call graph can be used to determine if it is possible to reach a machine-interacting library method from any of a machine type's event handlers. However, this is severely complicated if a library method also contains its own event handlers. There is no easy way to guess which machine's context (and therefore which machine type) will execute an event handler unless that handler is syntactically associated with the machine type.

Library-induced pessimism could be reduced by introducing new Machine Java rules for library code. For example, machine-interactive behaviour (requesting new machines and communicating with channels) could be restricted so that it can only be contained within a machine class (and nested classes within a machine class), or in non-static library methods of library classes that implement a `MachineAware<T,U>` interface where

the generic type parameters represent the types of machines that the library can interact with. Requiring machine-interactive library code to be non-static forces each machine to instance the library[4] with the appropriate type parameters for the machine's specific purpose for the library. The exploration of this concept remains future work.

Network-Chi uses the following two-part procedure to extract an application's pessimistic machine dependency graph:

- Some Java bytecodes have class dependencies. For example, the `ALOAD` opcode loads a reference from an object's field and the opcode has a dependency on the static type of that object. While generating the itineraries for an application's methods, each opcode is considered and a machine dependency has been found if:

  - The opcode's dependent class is a subclass of `Machine`.
  - **and**
    - The opcode is contained in a method in a subclass of `Machine`
    - **or** The opcode is nested and is eventually enclosed by a subclass of `Machine`.

  When a dependency has been found an edge is added to the dependency graph from the machine type that encloses the method containing the opcode to the machine type referenced by the opcode.

- The previous step yields a messy but acceptable graph of machine dependencies. This graph will include undesirable reflexive edges, edges between a machine and its superclasses, and edges between processor managers (which are also machines). The graph can be 'cleaned' by:

  - Removing all vertices that are defined by the Machine Java framework, leaving only application defined machines in the graph.
  - Removing all reflexive edges. From a code-generation perspective these edges do not provide any utility. Restricted versions of this graph that only capture communication may be useful for machine allocators, and in this case the reflexive edges may also be useful.

[4]Singleton patterns using a static field must also be prevented unless the implementation enforces inter-machine isolation of statics.

A machine classes' channels and their properties are somewhat easier to determine statically. Channels are determined by the public fields in a machine class with a static type that implements the `Intermachine` interface. As channel fields must be declared **`final`**, the exact properties of the channel can be determined by the object that is assigned to the field in the machine's constructor. A Machine's constructor (and field initialisers) are not permitted to interact with other machines or to perform processor local input/output. This means that a compiler or verifier can execute the machine's constructor and observe the channel objects that are instantiated, and there can be confidence that these observations will accurately predict the channels that an a machine constructs. The principle that a machine's constructors will always behave the same regardless of execution context is central to Machine Java's runtime operating mechanism, machine objects used as references to remote machines are guaranteed to have correctly matching channel objects (as they are the same type), and therefore will be able to generate compatible channel connectors when requested by application code.

5.3.1.3 Machines as Java Objects

The machine model is a particularly dynamic interpretation of actor oriented concurrency. The model does not provide a way to statically define the machine instances that will exist at runtime and Machine Java is even more restrictive as machine *references* are indistinguishable from machine *instances*; machines are essentially addressed by their own objects. This means that it does not make sense to have a machine reference to a machine that does not yet exist, just as Java cannot express a reference to an object that does not yet exist. This duality of object reference rules and machine reference rules makes machine oriented programming easier as machine objects do not have to be treated specially. Additionally, machines and ordinary Java objects have very similar rules that determine their lifetimes: a machine object used as a reference will only exist as long as it is reachable, just as any other Java object. A machine object that represents an *active* machine instance will exist for as long as it has pending or future possible behaviours (this topic was covered in more detail in section 4.3.5.). Importantly an application cannot determine if a machine object is a reference to a remote machine or an active instance hosted on the same processor.

Mechanisms have been devised to allow promises of future objects to be expressed (such as `java.util.concurrent.Future<V>`, or Scala's `scala.concurrent.Future` used

by Akka [91]) but these are intended to represent the eventual result of an asynchronous computation. Futures are not intended to express the future existence of a *specific* instance but rather the unknown result of a computation, therefore futures are not suitable for statically defining the runtime layout of an application.

5.3.1.4 Resource Consumption and Real-Time Guarantees

Machine Java's event driven architecture allows some efficiencies as it does not require an execution context such as an OS thread for each event source or handler. Even machines are not *required* to have their own parallel thread of control, and this enables implementations where machines are cheaply multiplexed onto each processor by sharing an event queue. Avoiding dependence on preemptive execution provides:

- A one-time reduction in binary size as considerably smaller 'kernel' is required.
- A potentially linear reduction in memory consumption for each machine as no memory is required to save the execution context of a machine on preemption.

However, these efficiencies only reduce the minimum required resource consumption of an application but they do not address whether or not an application's absolute requirements will exceed the capabilities of a specific platform.

The unknown runtime structure of an application presents a challenge for bounding the resource consumption of an application. If it cannot be known how many of each machine type will exist, then it also cannot be known how many resources will be consumed. A machine will consume *abstracted* resources (computation, memory and communications capabilities) without explicit declaration by the code, but the use of any non-abstracted resources (such as a serial port, or block device) must be asserted in the machine's constructor using the `Machine.REQUIRE_RESOURCE()` method. This makes the consumption of non-abstracted resources trivial to calculate if the multiplicity of the machine type can be determined.

Calculating the consumption of abstracted resources —memory and execution time in particular— is not nearly as straightforward, even assuming it is known what machine instances a processor will host. Standard Java has a number of features that complicate its analysis: concurrency provided via a flexible threading model, automatic memory management, and a polymorphic object model (enabling virtual dispatch and differentiated exception handling). The difficulty and necessity of determining the Worst Case

Execution Time (WCET) for real-time java applications has resulted in a considerable body of research [79]. The estimation of a Java application's worst case heap allocation has also been considered [175], and the Real-Time Specification for Java [73] (RTSJ) provides non-garbage collected memory models which are both more timing-predictable but also provide statically defined upper bounds on memory consumption. Safety Critical Java [203] (SCJ) further limits the memory model available to applications, and this enables even stronger analysis of Worst Case Memory Consumption (WCMC) [9]. The application of WCET and WCMC research to Machine Java is not considered in this thesis and remains potential future work.

Machine Java's construction does not directly address the design of real-time applications, and this is considered to be out of the scope of this thesis. However, the remainder of this section provides a brief discussion of some issues that would be relevant when considering a real-time machine-oriented application. Relevant details of the application model include:

- All machine instances are isolated so share no memory or resources, and cannot make assumptions about the execution rate of their code. This means that the execution of two machines sharing a single processor cannot be distinguished from the execution of machines on separate processors. This requires that the single-processor is time-sliced between the execution of the machines with a fine enough granularity. Assuming a simple and extremely fine-grained time slicing of processing resources, each machine can itself be considered to be a uni-processor for the purposes of scheduling its internal event handlers.

- Event handlers within a machine are non-preemptive, implying the use of non-preemptive scheduling and analysis techniques. A summary of non-preemptive uniprocessor scheduling results can be found in [66]. A significant difficulty in non-preemptive real-time scheduling is the high likelihood of priority inversion due to long-running low-priority tasks.

- Machines use a Highest Priority First (HPF) *non-idling* scheduler. Machine Java does not provide a direct mechanism to specify the deadlines for event sources, but if extended to provide some expression of deadlines then existing work can be used to determine priorities for the HPF scheduler. The use of a non-idling scheduler (the machine is never idle if there are pending events to handle) is even

more prone to priority inversion but finding feasible schedules for *idling* schedulers known to be NP-complete [66].

5.3.1.5 Forcing a Static Application Structure

In spite of the Machine Java's dynamic approach to application elaboration, it *is* possible to encode a static runtime layout of machine instances by using a rigid programming pattern that forces this result. Such a pattern enables models such as Synchronous Data Flow (SDF) [117] to be expressed. SDF models require a fixed number of data items to be received from the input channels each time the node (in this case machine) is 'fired', and when fired the node sends a fixed number of data items from its output channels.

An application can be considered to have a pre-elaborated static structure if it is possible at compile time to know exactly which machines will exist, and what their relationships to each other will be. The simplest way to achieve this is by ensuring that each machine type is instantiated at most once for the whole application (so that each machine instance can be considered to have its own type; an application-wide singleton). This can be enforced by creating new subclasses of `Machine` and `Start` to enable a compiler or verifier to check the adherence to this pattern:

StaticMachine classes would be used to for all machines in the application and would be forbidden to create machines or to receive machine references via communications channels during normal operation. `StaticMachine` would extend the `SetupableMachine` class and receive all of its required machine references via this mechanism. Neither of the restrictions are difficult to implement in the Machine Java framework: The processor's local **ProcessorManager** could refuse to accept requests for new machines from subclasses of `StaticMachine` and the communications nexus could deserialise received machine references as **`null`** if the destination channel is contained within a `StaticMachine` subclass.[5] A `StaticMachine` would accept a generic parameter that expresses the n-tuple of machine references the subclass of `StaticMachine` requires to operate.

`StaticMachines` should not perform any operations before they have received their references to their dependent machines, and this could be enforced by ensur-

[5]The `SetupableMachine`'s channel would not be subject to this restriction as it is not a subclass of `StaticMachine`.

ing they do not reimplement the `internal()` method.

StaticStart classes would act as the entry point to a statically elaborated application. A subclass of `StaticStart` must perform two operations in its `internal()` method:

1. Instantiate every machine instance in the application using the blocking `newMachine()` method. `StaticStart` machines can only request machines that subclass `StaticMachine`, and only during its `internal()` method.
2. Then explicitly 'setup' each `StaticMachine` instance using the `_setup` channel defined in `SetupableMachine`. As soon as the first communication with a `StaticMachine` is performed no additional machines can be requested. Only machine references can be communicated to the `StaticMachines` via their setup channels.

As long as the `StaticMachines` are application-wide singletons (rather than singleton machines on a particular processor), then the static machine dependency graph will match the runtime machine instance graph.

If a static application layout is insufficient and a static pattern of communications is also required, then SDF-style rules can also be enforced.

- A new subclass of `StaticMachine` is created, "`SDFMachine`", and subclasses of `SDFMachine` are required to define the `setup()` method inherited from `SetupableMachine` to instantiate an `SDFCombiner`.
- For simplicity, machines with at most two input channels and two output destinations are considered, but this scheme could be extended to any fixed number of inputs and outputs. The general concept is that the `SDFCombiner` handles the reception of data from its assigned input channels and only executes an application event handler when the expected number of input data items have arrived. The combiner also proxies the output data items to ensure that the application code has sent the correct number. A 2-in-2-out `SDFCombiner` would require nine parameters to its constructor:

  1. The first channel to receive data.
  2. The number of data items to receive from the first channel per activation.
  3. The second channel to receive data.

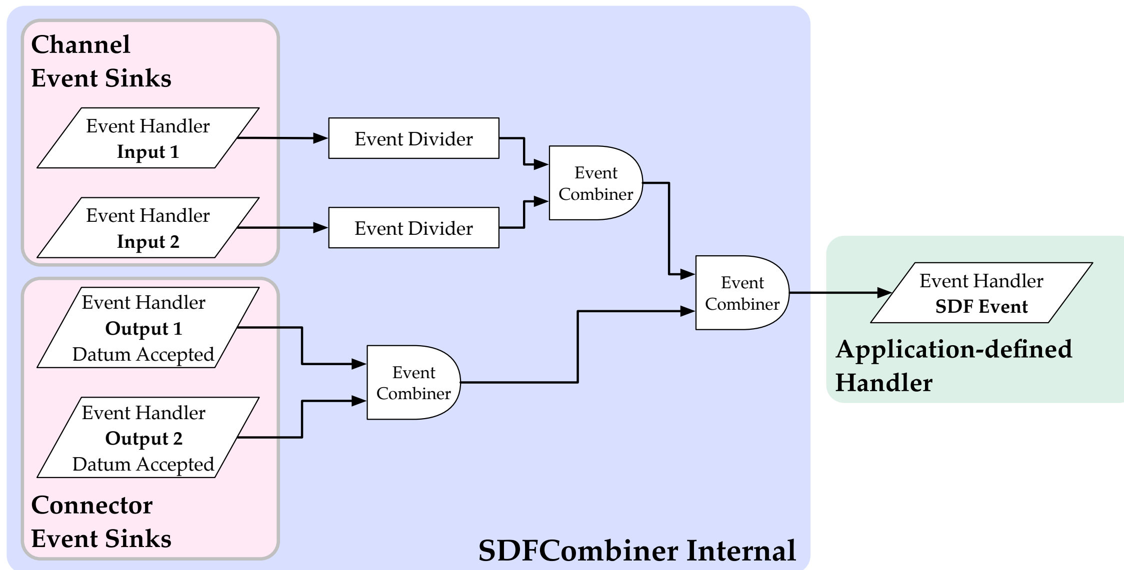


Figure 5.5: *The flow of events through a 2-in-2-out SDFCombiner. The combiner is an event source in a machine but is also a sink for four other event sources: the event handlers on the left are implemented by the SDFCombiner and service events from the input channels and output connectors if using non-destructive output protocols, such as* ***BoundedBuffer****s.*

4. The number of data items to receive from the second channel per activation.
5. A connector for the first destination channel.
6. The number of data items to send to the first destination per activation.
7. A connector for the second destination channel.
8. The number of data items to send to the second destination per activation.
9. An event handler that will be scheduled whenever sufficient input data has been received. This handler is expected to emit the correct number of data items to its two destinations. The machine is considered to have failed if the event handler returns before sufficient data has been sent.

The `SDFMachines` are sufficiently constrained that the execution of their `setup()` method is as predictable as their Java constructors. This enables the characteristics of their `SDFCombiner` to be determined by execution of the `setup()` method during compilation and observing how the machine choses to instantiate the combiner. This technique avoids the imposition of syntactic constraints on the construction of an `SDFCombiner`.

The internal architecture of an `SDFCombiner` can be seen in figure 5.5. An `SDFCombiner` acts as a sink for the input channel events and the 'data accepted' events for

```
1  public class EventDivider<T> {
2
3    private ArrayList<T> buffer;
4    private final Handler<? super Envelope<List<T>>> dividedHandler;
5
6    public EventDivider(TPIFUnidirectionalProtocol<T> source, int
         divide, Handler<? super Envelope<List<T>>>  dividedHandler) {
7      ...
8      source.setHandler(new Handler<Envelope<T>>() {
9        @Override
10       public void handle(Envelope<T> info) {...}
11     });
12   }
13 }
```

Figure 5.6: *An abridged definition of the `EventDivider`.*

non-destructive connectors. In turn, the combiner is a single event *psuedo-source*[6] that will trigger an event when sufficient data items have been received from the input channels and both output channels are ready to send again. As the `SDFCombiner` is not a machine itself it does not have an execution context nor its own event queue. The event handlers shown on the left of figure 5.5 execute in the context of the host machine, just as any event handler. However, the `SDFCombiner` is not a true event source as it invokes the machine's 'SDF event handler' directly during the execution of the most recent sink event handler. The event divider and event combiner are convenience utilities provided by Machine Java to ease the design of behaviour that depends on more than one event. Event combiners are discussed more thoroughly in section 5.3.3.1.

Event Dividers

The `EventDivider` class is a single event sink and pseudo-source used to aggregate data from unidirectional channels into fewer events. A `EventDivider` invokes the machine's supplied event handler with a `List` of the last n data items from the specified input channel. An abridged version of the `EventDivider`'s definition can be seen in figure 5.6. An `EventDivider` is parametric on the type of data item that it aggregates, and requires any unidirectional channel as a source of these data items. The application has to supply an event handler that is capable of accepting an `Envelope` of type `List<T>`

[6] *pseudo-sources* are false event sources in the sense that they do not have the capability to enqueue events onto the machine's event queue, but they do invoke event handlers so giving the appearance of an event source.

rather than just `<T>` as the channel itself requires.

5.3.1.6 Static Application Structure Summary

It has been shown that it is possible to design applications which will evolve a guaranteed runtime structure and that this structure easily be extracted at compile-time. This can be considered equivalent to a truly static description of machine instances but remains independent of an external description language. In addition the `SDFCombiner` provides runtime verification of the expected behaviour. However, a significant benefit of fixing an application's structure and communications quantities is the ability to use this information to generate a fixed communications schedule in advance of runtime. An implementation that is capable of translating a Machine Java application's behaviour onto a fixed runtime schedule is unlikely to require the complexity or validation capabilities of an `SDFCombiner` written in Java.

The ability to express Synchronous Data Flow applications should be interpreted as a demonstration of Machine Java's (and by extension, the MAA application model's) flexibility and not as a suggestion of an appropriate use-case. Synchronous Data Flow was originally described as a model for describing digital signal processing algorithms such that a parallel hardware implementation could be accomplished. If a DSP algorithm were implemented in Machine Java at the level originally intended for SDF the result would probably be inefficient, unnecessarily circuitous and a poor match for the strengths of the application model and Machine Java.

5.3.2 Application Platform Independence

All Machine Java applications will execute correctly and without modification on any platform if:

- An implementation of Machine Java's internal APIs is available that can support the processors and abstracted resources (memory and communications) that the platform specified.
- **and** the application does not use any non-abstracted resources.

The basic claim for the platform independence of applications is validated by the execution of many example applications (including the microbenchmarks detailed in section5.4.2.3) without modification on three substantially dissimilar architectures: A stan-

dard Java virtual machine, as a set of processes on a POSIX (Linux) OS, and directly on the embedded processors of the example Blueshell platform (discussed earlier in section 5.2.2).

In this context an application is considered to execute correctly if all of:

- The behaviour of its event handlers is consistent with the expected behaviour of the Java code that defines them.
- The behaviour of intramachine event sources and event scheduling is consistent with the descriptions in chapters 3 and 4.
- The intermachine interactions and isolation is consistent with the descriptions in chapters 3 and 4.

The application is considered to be the specification of its behaviour and therefore any encapsulating specifications (such as the application programmers *intended* behaviour) cannot be considered.

Unfortunately only trivial applications are capable of meeting the requirements for platform independence. Interaction with any external entities can only happen via non-abstracted resources and therefore introduces an element of platform dependence. Even highly limited interactions such as debug logging are not fundamentally abstracted by the application model, even though Machine Java supports this activity in the `Machine` class. The use of `Machine.log()` introduces an implicit requirement on a debug logging resource which is not universally required by the platform model. Any platform that does not provide such a debug logging capability will not be able execute an application that uses `Machine.log()` according to the correctness criteria defined above. This is not a practical problem if the divergence from the specified behaviour is expected.

Non-abstracted resources in the platform model (subclasses of `Resource` in Machine Java) are only tokens to represent the existence of some device or capability, and the same token is used by an application to indicate utilisation of the represented entity. Non-abstracted resources are a mechanism to introduce constraints on the platforms that are capable of executing the application, but they do not necessarily couple the application to a single specific platform. However, as the procedure required to use a resource is unspecified both by Machine Java and the MAA models, the use of a highly domain specific resource (such as an attached peripheral) could result in the effective coupling of an application to a specific platform.

Even if non-abstracted resources are excluded from consideration, there are several situations in which an application can benefit from platform specific information that is not (and *cannot* generally be) provided by Machine Java or the application model. Examples of useful platform dependent information includes:

processor count Applications that have large quantities of easily subdivided work (so-called *Embarrassingly Parallel* problems) will likely execute most efficiently when there is a machine contributing to the task on each processor of a platform. Allocating too few machines risks underutilisation of the platform and too many machines will result in an increased proportion of processor effort being spent on the management of machines rather than useful work. However, applications cannot directly determine the number of processors available at runtime nor the loading of each processor. Java does provide a basic facility to determine the number of *logical* hardware threads currently available via the `Runtime.getRuntime().availableProcessors()` API but this is not supported in the current formulation of Machine Java, and the platform model does not guarantee that any facts about the complete, global platform are knowable at runtime. Each processor in a platform may only have a awareness of other processors within a certain local area. Even if an application can determine the number of idle or under-utilised processors there may be other factors that affect the wisdom of allocating more machines in an attempt to achieve higher throughput. Machine Java cannot elegantly express non-functional intentions for machine allocations (such as "only if this would increase throughput", or "if there is spare power") and this is a clear opportunity for further research. Some non-functional objectives, such as power-awareness of machine allocation, can be crudely emulated via processor resource availabilities and machine resource requirements.

available memory Applications may be able to take benefit from using additional memory if it is available, but also function correctly if it is not. Java's *soft* references provide a mechanism for applications to use memory optimistically. In standard Java objects that are only soft-referenced will be garbage collected before an `OutOfMemeoryError` is thrown. This mechanism would only work within a single machine, is not supported by Network-Chi, and does not address machines which can optionally exist but have high necessary memory requirements when they do exist.

Machine Java does not allow a machine to be conditionally requested depending on the availability of desired but not required memory; the construction of a new machine is necessarily conditional on the host processor meeting the memory *requirements* of a new machine.

power and thermal constraints The electrical power consumption and heat output of a platform will often be dependent on the activities of the software being executed. Machine Java does not have a way for an application determine the acceptable power or thermal constraints, nor a way to react to changes in these constraints. This is a particularly important limitation for platforms where power and thermal constraints are not enforced in hardware but can be violated by software activity. Such platforms can only be adequately addressed by introducing dependencies on the relevant sensors and regulators into the application, but even this may be difficult if the platform's constraints are more complex, such as where the power constraint varies as a function of time across multi-processor region.

5.3.3 Complex and Dynamic Application Structures

In section 5.3.1.5 the ability of Machine Java to express static application structures was considered but the MAA application model does not require an application's runtime structure of machine instances to be statically extractable, and therefore Machine Java does not either; Machine Java applications have the freedom to dynamically request machine instances. The complexity of a machine's event handlers is only limited by the specification of the Java language and Machine Java does not impose additional constraints on when a new machine can be requested via one of the `newMachine()` methods. This means that new machines can be requested in response to stimulus from other machines, from interaction with resources, or according to the results of any expressible computation.

Applications can contain any number of machine instances at runtime and the relationships between an application's machines can form arbitrarily complex topologies. Familiar patterns of parallel computation from fine-grained pipeline architectures to coarse-grained replication of tasks (such as in a client-server arrangement) can all be modelled using actor orientation and so are all implementable in Machine Java. Naturally, other more exotic arrangements including rings, grids, toroids, cubes, hypercubes, trees and entirely irregular arrangements of machines are also expressible in Machine

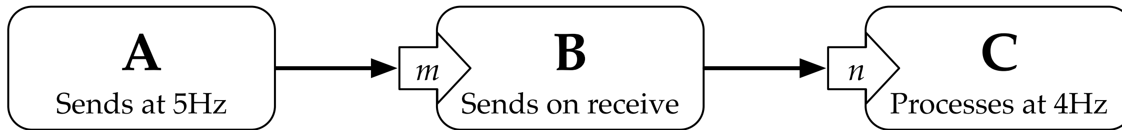


Figure 5.7: *An example application with three machines A, B, and C that are formed into a pipeline arrangement. Regardless of B and C's channel buffer lengths, m and n, messages will always eventually be lost as C cannot process data fast enough and B does not control the rate of data generation.*

Java.

Although complex arrangements of machines can be constructed at runtime in Machine Java, coordinating communication within a machine is not necessarily straightforward. As explained in section 3.2.2.1, all machines are mutually asynchronous and any temporal relationship established between a pair of machines only exists as long as it is actively maintained via communication. This asynchrony complicates even simple architectures where a machine, *B* will send data to another machine, *C*, in response to the receipt of a message from a third machine *A* that is unrelated to *C*. This arrangement is illustrated in figure 5.7. If it is acceptable for messages to be lost then there is no issue: destructive channel protocols such as **OverwritingBuffers** can be used by machines *B* and *C*. If it is unacceptable for messages to be lost then machine *A* must be slowed down to accommodate *C*'s inability to process data as fast as *A* can produce it.

Using non-destructive communications channels (such as a **BoundedBuffer**) in *B* and *C* will ensure that no data is lost during operation but machine *B* would eventually crash (due to a `PreviousDatumNotAcceptedException` exception (see section 4.4.1.1) being thrown from the non-destructive buffer connector) when it attempts to forward a datum when *C*'s buffer is already full. As all send operations in Machine Java are non-blocking, the use of non-destructive channels alone is not sufficient to produce a functional pipeline. Application code must always wait for the framework to *accept* the most recently sent data item before attempting to send another, and data is only accepted by a channel if there is an available slot in its receive buffer for the item.

5.3.3.1 Event Combiners

In the example shown in figure 5.7, machine *B* can effectively rate limit *A*'s production of messages by only reading from its channel when *C* can be sent to. Data is received from a channel's internal buffer only when the `Envelope` provided to the channel's event

```
1  public class EventCombiner<A, B> {
2    public EventCombiner(Handler<? super Pair<A,B>> combinedHandler)
         {...}
3
4    //Handlers for input event sources
5    public Handler<A> getLeftHandler() {...}
6    public Handler<B> getRightHandler() {...}
7
8    //Preset the inputs:
9    public void rightSideReady() {...}
10   public void leftSideReady() {...}
11 }
```

Figure 5.8: *An abridged definition of the* `EventCombiner`*. An event combiner executes an application defined handler when both input event handlers have been invoked since the combiner was last triggered.*

handler is 'opened' by calling it's `getPayload()` method. Therefore application code is able to delay the receipt of data items by delaying its call to `Envelope.getPayload()`. This is further complicated by the non-preemptive execution of event handlers. In the situation where B is handling a data received event, but the onwards channel to C is not yet ready to accept another send, B must save the unopened `Envelope` in a **`private`** data structure and return from the event handler to wait for the 'data accepted' event handler to be invoked. Alternatively B could repeatedly attempt to send the new data item to C, each time catching the `PreviousDatumNotAcceptedException` and trying again until successful, but this behaviour results in the starvation of all other event sources in B. Using the `getPayload()` deferral strategy it is possible for machines to create back-pressure that will prevent other machines from overwhelming it with data, while also remaining responsive to other event sources.

If machine A uses a periodic event source to trigger the transmission of data to B then it will also require an event deferral strategy. In this case it should only send when the channel to B is ready and when timed event has also occurred. The requirement that some behaviour is dependent on more than one event occurs so frequently in Machine Java applications that it justified the creation of a general purpose *Event Combiner*.

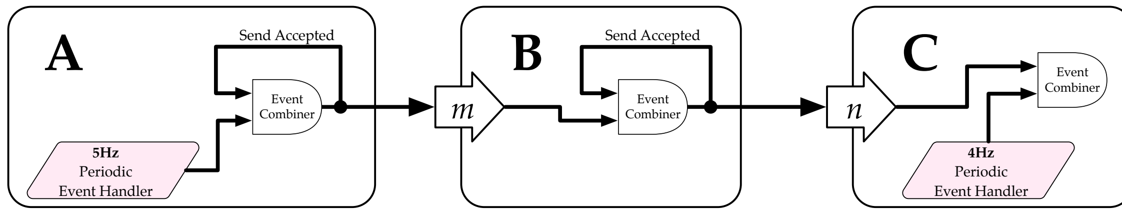


Figure 5.9: *Event combiners enable the application in figure 5.7 to proceed only as fast as the machines and buffer space will allow.*

5.3.3.2 Time Driven Behaviour with Event Combiners

Time driven behaviour fits naturally into Machine Java's event-driven programming style, even when intended behaviour is dependent on multiple prerequisite events. The `EventCombiner` class provides a clean and type-safe mechanism for triggering application code only after multiple events have occurred. The abridged definition of an event combiner can be seen in figure 5.8: Combiners provide two event handlers that can be supplied to any other event source in the same machine. As soon as both of the combiner's input event handlers have been executed the combiner will invoke the handler supplied to its constructor. This handler must accept a `Pair<A, B>` where `A` and `B` match the type parameters for the 'left' and 'right' side of the combiner, respectively. Figure 5.9 shows how event combiners could be used to ensure that each machine only sends data when appropriate. In this example, machine *A*'s event combiner would require a handler with the type: `Handler<Pair<TimeEvent, Nothing>>` and would need to have the 'data accepted' side *preset* immediately after construction as connectors can only trigger a data accepted event after the first data item has been sent.

An example of a more complex arrangement of event combiners is used in the implementation of the **LevelSensor** machine (see figure 3.2), and these are shown in figure 5.10. The **LevelSensor** machine uses two combiners to execute a handler only after *three* event sources are ready. Any number of event sources can be combined by chaining event sources as in figure 5.10 but it can be seen that even with only three events combined the handler's type signatures are becoming very large. The application's ultimate event handler will have as many `Pair<A,B>` containers in its type signature as the number of event combiners that were used to trigger the composite handler. The Java compiler is able to verify that the final event handler's use of its parameters is consistent with their types, but if two or more of the input event sources share the same parameter type it is possible (and quite plausible) for a type-safe confusion of the event parameters

```
...
//A combiner that fires when both sensors are ready and when the
    timed interval has happened.
private EventCombiner<TimeEvent, Pair<Envelope<Integer>, Envelope<
    Integer>>> timedPollCombiner
  = new EventCombiner<TimeEvent, Pair<Envelope<Integer>,Envelope<
      Integer>>>(
    new Handler<Pair<TimeEvent, Pair<Envelope<Integer>,Envelope<
        Integer>>>>() {

  public void handle(Pair<TimeEvent, Pair<Envelope<Integer>,
      Envelope<Integer>>> info) {
    //retrieving the values from the pairs prone to errors:
    int sensorAValue = info.right.left.getPayload();
    int sensorBValue = info.right.right.getPayload();

    //Calculate level as the mean of the inputs:
    currentTankLevel = (sensorAValue+sensorBValue)/2;

    if (currentTankLevel>EMERGENCY_THRESHOLD) emergency.openValve.
        signal();

    //query the sensors again:
    querySensors();
  }
});

//Combine the two sensor's data arrivals into one event:
private EventCombiner<Envelope<Integer>, Envelope<Integer>>
    sensorResponseCombiner = new EventCombiner<Envelope<Integer>,
    Envelope<Integer>>(timedPollCombiner.getRightHandler());

protected void internal() {
  ...
  //A periodic task to poll the sensors: (event combined with
      sensor responses)
  new Period(POLL_MILLIS, timedPollCombiner.getLeftHandler());
}
...
```

Figure 5.10: *An excerpt from the* ***LevelSensor****'s implementation. This machine uses two event combiners to determine when to update it's* `currentTankLevel` *and to check for an emergency condition.*

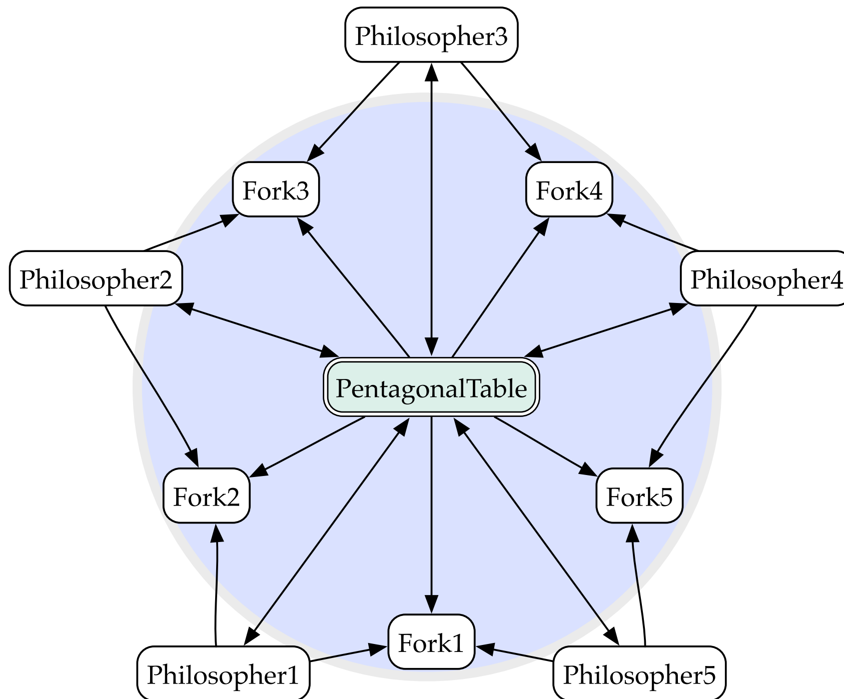


Figure 5.11: *The arrangements of the dining philosophers at a table. This is the static machine type dependency graph for an application where each philosopher and fork was a distinct Java class. All rectangles represent a machine type and* `PentagonalTable` *is the application's start machine.*

to occur. A full listing for the example **LevelSensor** implementation is reproduced in appendix B.3

5.3.4 Concurrency Without Sharing

A major consequence of the application model is that there are no situations in which locks, semaphores, monitors or any other mutex primitives are required. This is because the model provides no opportunities for concurrent access to the same data structures. Data within a machine can be shared between event handlers, but a single machine's event handlers never execute concurrently. Each machine executes concurrently with respect to all other machines but no two machines can have access to the same data or resources.

As there is never shared concurrent access to data, the *producer-consumer* and *reader-writer* problems do not occur. In particular the producer-consumer problem (where two processes intend to communicate via a bounded buffer stored in shared memory) is not only inexpressible in an machine-oriented context but communication via a bounded

```java
1 public class Fork extends Machine {
2   protected Philosopher owner;
3
4   public final RemoteProcedureCall<Nothing, Boolean> gainFork = new
        RemoteProcedureCall<Nothing, Boolean>(1, new Handler<
        ReturnableEnvelope<Nothing, Boolean>>() {...});
5
6   public final BoundedBuffer<Philosopher> releaseFork = new
        BoundedBuffer<Philosopher>(1, new Handler<Envelope<Philosopher
        >>() {...});
7
8   protected void internal() {
9     //Forks have high priority compared to philosophers and the
          release() has higher priority than gain()
10    setPriority(5);
11    releaseFork.setPriority(5);
12  }
13 }
```

Figure 5.12: *An abridged reproduction of a Fork machine. The* `owner` *field records a reference to the current philosopher that has control of the* `Fork`*, or* **`null`** *if the* `Fork` *is idle. The highlighted statement is essential to prevent* live-lock *of the application when executed on a single processor platform. This class can be found in full in appendix C.4.1*

buffer is explicitly addressed by the application model.

5.3.4.1 Dining Philosophers in Machine Java

A third common problem in concurrent computing is the so-called *Dining Philosophers Problem* [83]. This problem is not about correctly coordinating processes that intend to use shared memory, but it is a broader problem about sharing resources between concurrent entities. The dining philosophers problem is about ensuring the behaviour of five hungry philosophers such that they do not starve in an environment where two forks are required to eat but there are only five forks in total. The philosophers are typically seated at a round table with a fork in between each philosopher. Each philosopher can only eat using their two adjacent forks. This arrangement can be seen in figure 5.11, which is the extracted machine dependency graph of a Machine Java application.

The dining philosopher's problem is an instructive example for several reasons:

- In this problem the forks are effectively a shared resource that needs to be ac-

```
newMachine(Fork1.class, new Handler<Fork1>() {
  @Override
  public void handle(Fork1 info) {
    fork1 = info;
    newMachine(Fork2.class, new Handler<Fork2>() {
      @Override
      public void handle(Fork2 info) {
        fork2 = info;
        newMachine(Fork3.class, new Handler<Fork3>() {
          @Override
          public void handle(Fork3 info) {
            fork3 = info;
            newMachine(Fork4.class, new Handler<Fork4>() {
              @Override
              public void handle(Fork4 info) {
                fork4 = info;
                newMachine(Fork5.class, new Handler<Fork5>() {
                  @Override
                  public void handle(Fork5 info) {
                    fork5 = info;
//Create our philosophers!
setupMachine(Philosopher1.class,
      p(1, p(PentagonalTable.this, p(fork1, fork2))),   null);
setupMachine(Philosopher2.class,
      p(2, p(PentagonalTable.this, p(fork2, fork3))),   null);
setupMachine(Philosopher3.class,
      p(3, p(PentagonalTable.this, p(fork3, fork4))),   null);
setupMachine(Philosopher4.class,
      p(4, p(PentagonalTable.this, p(fork4, fork5))),   null);
setupMachine(Philosopher5.class,
      p(5, p(PentagonalTable.this, p(fork1, fork5))),   null);
                  }
                });
              }
            });
          }
        });
      }
    });
  }
});
```

Figure 5.13: *An excerpt from the `internal()` method of the `PentagonalTable` machine. Even with only five `Philosophers` and `Forks` the setup is quite convoluted. Note: `p` constructs a `Pair` from its two arguments.*

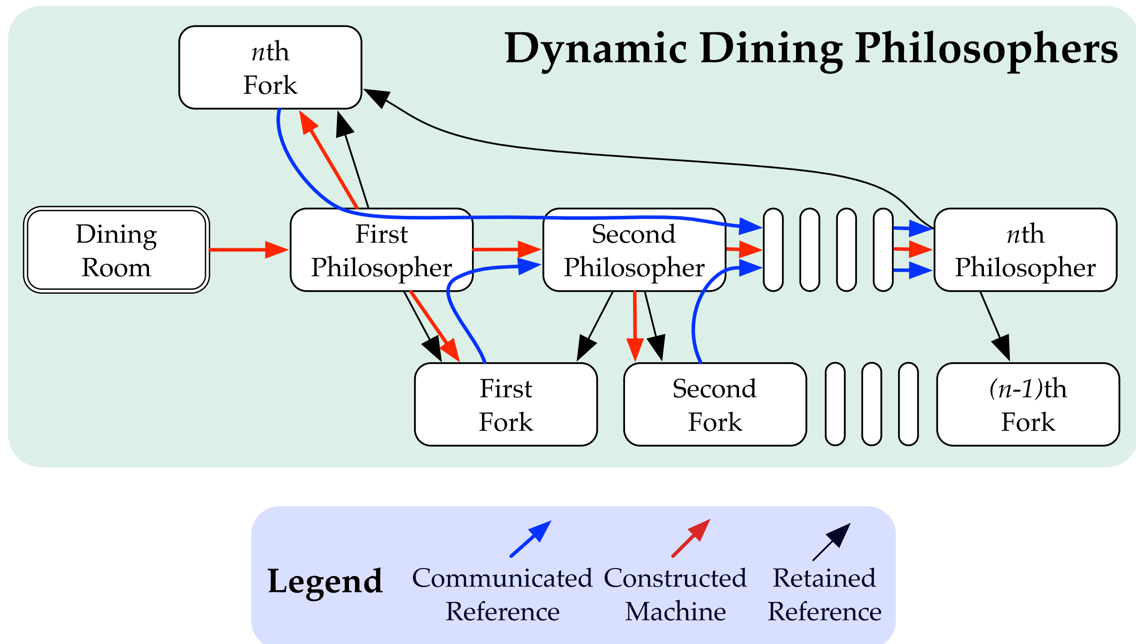


Figure 5.14: *The elaboration process for the dynamic dining philosophers application. The first* `Philosopher` *constructs both of its* `Forks` *and the last* `Philosopher` *constructs none.*

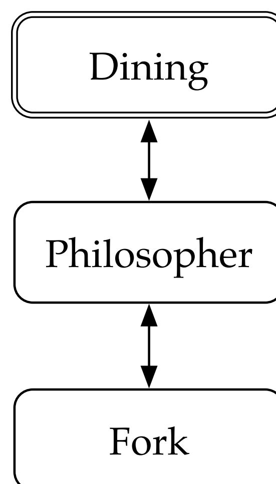


Figure 5.15: *The static machine type dependency graph for the dynamic dining philosophers application. Contrast with the application structure shown in figure 5.11.*

cessed by more than one machine. 'Shared'[7] resources are well represented as their own machine, but there is also no choice: The only entity in an application that can hold state and be accessed from multiple machines is another machine. The representation of a fork as a machine can be seen in full in appendix C.4.1, with the important details illustrated in figure 5.12. A `Fork` exposes two channels to philosophers: a **RemoteProcedureCall** used by a philosopher to acquire a fork for their exclusive use, and a **BoundedBuffer** that is used to notify the `Fork` that it is no longer in use. Only the philosopher that currently has ownership of a fork is allowed to release it.

- During execution its desirable for each `Philosopher` machine to report each time it has been able to acquire its necessary `Forks` and has eaten. The natural approach is to log a message via `Machine.log()` but this mechanism is not safe for use on multiple machines simultaneously: It just maps to Java's `System.out` but with additional machine-specific information. When executing on a JVM this is not *too* problematic as logged messages from all machines will be interleaved and logged to the console. However, Network-Chi does not provide atomic console access so simultaneous logging results in unintelligible output as messages can become interleaved at a character level of granularity. The dining philosophers problem is a parable for the use of shared resources, and the debugging log is certainly a shared resource. The solution is to use the start machine (`PentagonalTable` in figure 5.11, and `Dining` in appendix C.4) as the arbiter of Machine Java's logging capability. All philosophers are issued with a reference to the 'dining room' machine and send messages to this machine which will log to the console on their behalf. Attention must be paid to the message rates to the logging arbiter and the buffer size of the arbiter's channel as message logging can easily become a performance bottleneck.

- The implementation of the dining philosophers problem in Machine Java highlights the differences between a static application structures, such as the one shown in figure 5.11 and more dynamic application constructions. The static application structure shown in figure 5.11 has advantages for resource planning and a clarity of structure but it introduces significant code duplication and requires more programmer effort to initialise the application: the programmer must explicitly

[7]In the sense that multiple machines require access.

instantiate each machine individually. An excerpt of the application initialisation from the `PentagonalTable` machine is shown in figure 5.13. It is clear that this approach to application description does not scale programmer effort gracefully; it is a non-trivial amount of work to change the number of philosophers: Each `Philosopher` class has a type signature which encodes which of the `Forks` it can accept. For example:

```
public class Philosopher1
  extends SetupableMachine
    <Pair<Integer, Pair<PentagonalTable, Pair<Fork1, Fork2>>>>
```

A much more elegant construction of the dining philosophers problem in Machine Java uses only one class for all philosophers and one for all forks. The dynamic dining philosophers application is reproduced in appendix C.4. The benefits of this construction are significant:

- There are fewer classes and therefore a smaller output code size.
- There is no redundant replicated code, so the application is much easier to update if the behaviour of forks or philosophers is changed.
- The number of `Philosopher` machines at runtime can be configured trivially by altering a constant in the `Dining` class.
- Programmer errors in the assignment of *specific* forks to philosophers (lines 22-31 of figure 5.13) are impossible as the assignment performed algorithmically.

In this construction each the application builds itself at runtime. The `Dining` machine creates the first `Philosopher`. Each `Philosopher` constructs the `Fork` required and the next `Philosopher` that will share one of its `Forks`. This application elaboration procedure is illustrated in figure 5.14. The dynamic elaboration of the philosophers does have two notable disadvantages: the extractable application structure is quite minimal (shown in figure 5.15) and it requires much more effort to understand and reason about the elaboration process.

- Finally, the dining philosopher's problem exposes more challenges of designing reliable applications in a machine oriented style, and with Machine Java in particular. One such challenge is avoiding implicit dependences on event ordering

across multiple machines. The application model makes it clear that only the event ordering within a machine is specified, so if an application makes assumptions on the ordering of events in separate machines then it is possible it will fail.

The philosophers in these example applications use a basic strategy for attempting to eat:

1. Attempt to acquire the first fork, if this fails try again.
2. While owning the first fork, attempt to acquire the second fork, if this fails try to acquire the second fork again.
3. While owning both forks, eat for a duration.
4. Release both forks and repeat from the start.

This procedure avoids deadlock as all philosophers share the same total ordering of forks; all philosophers will attempt to gain the least recently instantiated fork first. However, this procedure does make a subtle assumption about intermachine event ordering and is therefore faulty. As the procedure releases the forks and begins the procedure again immediately, there is an assumption that the forks will have been released and will be available for acquisition by other philosophers before this philosopher requests the fork again. This appears to be a reasonable assumption as the fork may have a queued acquisition request from its other philosopher and this can be assumed to have an equal chance of success to the local philosopher. However, in the case of a single processor platform Machine Java has entirely predictable and deterministic intermachine[8] event ordering and the ordering does not agree with this assumption. The following happens on Network-Chi targets:

1. Another philosopher sends a request for this philosopher's first fork. The event for this message is queued.
2. The philosopher finishes eating.
3. The philosopher sends a 'release fork' message to both forks.
4. These messages create events which are appended to the *processor*'s event queue.

[8]All machines share an event queue on the same processor when using the Network-Chi implementation.

5. The philosopher starts its eating cycle again and sends a request to the first fork.
6. The event for this message is appended to the processor's event queue.
7. All events in the queue are handled before the release fork messages are processed. So the request issued by the other philosopher in step 1 is handled and rejected as the fork has not yet been released.
8. The release messages are processed and the forks are released.
9. The next event in the queue is this philosopher's request for the first fork which succeeds.
10. The philosopher continues its procedure to eat, which will succeed.

This sequence of events will repeat forever resulting in a live-lock of the application. A philosopher will always be able to eat repeatedly and another will be starved.

This problem can be addressed in the uniprocessor case by increasing the priority of `Fork` machines, and increasing the priority of the `Fork`'s `releaseFork` channel. This is highlighted in figure 5.12. However, more complex protocols are necessary to ensure application-wide liveness on a realistic platform.

5.3.4.2 Deadlocks in Machine Java

In addition to the live-locks that are possible in any distributed system, Machine Java provides a facility that can result in network deadlocks on some platforms. There is only one blocking operation provided in Machine Java, for the creation of a machine via the `newMachine()` method, but this is sufficient to deadlock a simple network if it is not used carefully. Any application where more than one machine will construct other machines is vulnerable to deadlock if the platform's internal API implementation uses non-preemptive multiplexing of machines to processors. Machine Java's implementation on Network-Chi uses non-preemptive machine multiplexing, so this is not only a theoretical concern.

As a concrete example, consider a basic platform with two processors that multiplex machines non-preemptively. A processor in this platform is unable to service requests to create new machines while it is blocked waiting for a remote processor to create a

machine for it. If an application loops creating machines which in turn also create machines then a deadlock becomes highly likely. The cause of the deadlock is that both of the processors become stalled waiting for the other processor to service their request for a new machine. If the network media is capable of re-ordering interprocessor messages and the Machine Java internal management messages (in this case `MachineControl` and `MachineControlResponse`, see section 4.4.4) are reordered from processor B, then deadlock becomes guaranteed without even repeated machine requests on processor A.

Machine model applications have no way to specify where machines should be allocated at runtime, so there is no way that a developer can encode constraints to a `newMachine()` request that would prevent the deadlock described here. The most universal solution is to avoid using the blocking version of the `newMachine()` API in favour of the event-driven `newMachine()` and `setupMachine()`. If an application is simple enough that only the start machine requests other machines, or where the application is structured to avoid simultaneous machine requests, then it can safely use the blocking `newMachine()` API. This API has the advantages that it is simpler to use and it can throw exceptions if there was a problem requesting the new machine.

5.3.5 Fault Tolerance in Machine Java

Java's abstraction of memory and processing easily leads to application design with the implied *fault hypothesis* [112] that there will be no failures in application-visible memory or execution. Within the context of a single event handler, Machine Java does not provide any substantial additional benefits for fault tolerance. However, machines are not invalidated if an event handler crashes by allowing an exception to propagate out, and in this sense machines can be considered to be more resilient execution contexts than standard Java threads. If an exception propagates out of the method started by a standard Java thread then the thread is *abruptly* terminated. The ability of a machine to survive a failed event handler does not imply that the machine's data structures are consistent, therefore the additional *resilience* of a machine compared to a thread is not necessarily additional fault tolerance.

The strong isolation of machines does have implications for the fault tolerance of whole applications: The failure of a machine cannot render the internal state of any other machine inconsistent. Machine Java does not explicitly provide fault tolerance to applications, but does ensure that failures of application code do not automatically

propagate between machines.

Interaction between machines via non-destructive channels, such as a **BoundedBuffer** or any bidirectional channel type, establishes a synchronisation relationship between the machines. For the sender machine this relationship exists only between the transmission of the 'request to send' message and the receipt of the 'clear to send', but even this transient synchronisation relationship introduces vulnerability to failure if the receiver machine is defunct. All ordinary intermachine communication is non-blocking in Machine Java, so an attempted send to a defunct machine via a non-destructive channel connector only implies the connector becomes unusable as the datum is never successfully sent; the machine will continue to execute as normal. Machine Java does not provide any mechanism to cancel a send to a non-destructive channel so unusable connectors represent a permanent loss of memory. Machines that interact using non-destructive channels can accommodate fault hypotheses with bounded transmissions to failed remote machines.

Destructive communications channels (such as the **OverwritingBuffer**) do not require a handshake between machines so allow fault hypotheses with unbounded numbers of transmissions to failed remote machines. In both destructive and non-destructive cases, application code can use timed event sources (described in sections 3.2.3.2 and 4.4.3) to detect potential failures of remote machines.

In the case of the total hardware failure of a NoC processor, it would be possible for the failed processor's local router to automatically reject requests for new machines. This is because the `MachineControlMessage` class used by Machine Java's `Processor-Manager` to request machines has a fixed size and a predictable serialised layout. A router, when in 'failed processor' mode would only have to determine that the network packet is a Java object and detect that the object is a `MachineControlMessage` via the type identifier present in byte indices 1-4 of Network-Chi's on-network object format. The `MachineControlResponse` message used to indicate a successful or failed machine creation also has a fixed size and layout, and this would enable a router to automatically respond with a hardcoded but valid `MachineControlResponse` to indicate that the machine could not be created. A serialised `MachineControlResponse` is 76 bytes in length and so it would require ten LUTs of a Xilinx Virtex 7 architecture FPGA[9] to embed a fixed failure response message into a network router.

[9]Every Virtex-7 series LUT can implement a 1×64bit ROM [229].

5.3.5.1 Handling Failure in Machine Java

It is a limitation of this formulation of Machine Java that it does not provide any high-level facilities to assist with handling failure. Failures of hardware (such as processor or network failures) and failures in application code can be tolerated at a basic level by implementing communications protocols on top of the destructive channel protocols. Significant architectural work is required to build a resilient application using destructive channels: no guarantees of message delivery or notification on failure are provided by the framework with these channels. The Machine Java channels are primarily designed to guarantee the memory safety (the impossibility of buffer overruns) in receiver machines.

Machine Java also does not provide any built-in mechanism to detect the failure of remote machines automatically. It is possible to introduce Erlang-style failure notification semantics into Machine Java by introducing a new class of Machines for this purpose: A `ReliableMachine` class could extend `SetupableMachine`, accepting a reference to another `ReliableMachine` as its setup parameter. The machine supplied as the parameter would be the reliability 'monitor' or 'parent' of the new machine. The `ReliableMachine` class would predefine a non-destructive **OverwritingBuffer** channel 'machineFailure' which would be sent a message on the event of a failure in the event handler in one of the machine's dependent `ReliableMachines`. To ensure that failures in a `ReliableMachine` 's event handlers are propagated to it's parent machine, the `ReliableMachine` should also implement an event-handler factory: Usually application code directly instantiates instances of `Handler<...>` to handle events, but `ReliableMachine` event handlers should be requested from the machine's own handler-factory that intercepts exceptions thrown by the machine's event handlers and then forwards them to the parent machine. The handler factory would produce `ReliableHandler` objects. The Machine Java runtime could also be extended to enforce this event-wrapping mechanism by only permitting `ReliableMachines` to enqueue events that are handled by subclasses of `ReliableHandler`.

`ReliableMachine` classes are not inherently fault tolerant but would permit more reliable applications by allowing failure notifications to be automatically propagated between machines. Even more complex schemes such as automatic machine failover or robust n-modular redundancy could be implemented with more layers of abstraction on top of `ReliableMachine`.

5.4 Overheads and Scaling

In chapter 3 the claim was made that *machine oriented* applications would be scalable, portable and capable of execution on resource constrained platforms. In order to evaluate these claims it is necessary to have a realisation of the model, and chapter 4 presented Machine Java, a Java-language realisation of the Machine Abstract Architecture. Machine Java also furthers this thesis' overall objective of demonstrating that the use of an actor oriented programming model enables non-cache-coherent multiprocessor networks on chip to be programmed with a familiar general purpose programming language.

Having already considered a number of evaluation criteria in the previous section, this section addresses the following evaluation criteria related to the overheads and potential for scalability:

- What are the static overheads of a machine Java application (how much read-only memory is required for a Machine Java application) and to what extent are these reasonable? This is considered in the next section, 5.4.1.
- What is the overhead for communication between machines at runtime, and does this indicate scalability? This is considered in section 5.4.3.
- What are the computation overheads of the Machine Java runtime, and does this also indicate scalability? This is considered in section 5.4.4.

5.4.1 Static Memory Consumption

The *static* memory consumption of an application refers to the absolute minimum requirements for read-only memory. Applications compiled with Network-Chi require read-only memory for program code and string literals. As this memory is read-only it can be safely shared between multiple processors without any implications for the functional correctness of an application. The coherence or existence of processor-local caches has no impact on the execution of an application, but these factors can certainly impact non-functional characteristics including performance, timing and power consumption.

As discussed in section 5.3.1.2 there is no requirement that every processor on a platform has access to the same program code. Each processor only needs to have access to program code for the framework and the machines that are planned to be

allocated to the processor. Ideally the platform implementation on each processor will have some awareness of the capability of processors in its local area to host each type of machine, but this is not required. If a machine is requested from a processor without access to the requested machine type's code then the creation will fail and a failure `MachineControlResponse` will be returned to the requesting machine. The dynamic machine constructor (*EssentialReflection.createMachineDynamic()*, see section 4.6) used by a processor's `ProcessorDriver` to instantiate machine objects at runtime will return **`null`** if the executing code was compiled without support for the specified machine type.

In this section the figures are for binaries that include support for all machines in the application. As the Network-Chi compilation workflow only includes code that is potentially used it is not easy to calculate the impact that excluding a machine will have on static memory consumption. For instance, some classes are likely to be used by multiple machines, while other functionality that would be expected to be shared (such as framework code) may only be used by a single machine. However, Network-Chi exposes a command-line switch to exclude named machine classes from the compiled binary. It is easy to support this workflow within Network-Chi: it considers excluded machines to be uninstantiated classes and therefore they do not qualify as live code for the purposes of code generation.

Including code within an application to support experimental data collection can have a considerable confounding effect on the static memory consumption if it depends on Java libraries that were not already used by the application.

There are several factors that influence the overall static memory consumption of an application:

- The application itself, the libraries that it depends on, and their transitive dependencies.

  Table 5.3 summarises the twenty largest contributions to the `SpeedTest` (see section 5.4.2.3) microbenchmark's static memory consumption. It can be seen that the string literals used by the application are the single largest contributor to static memory consumption, followed in a distant second place by the virtual method dispatch functions. Only one out of the twenty largest memory consumers is defined by the application. Nine of the contributors are standard Java libraries, either compiled unmodified or *replaced* by Chi to improve suitability for embed-

Component	Origin	code	(bytes) data	total
string literals	all	9,310	26,600	35,900
method dispatchers	all	13,000	0	13,000
`SimplerClass`	Replacement	5,620	2,330	7,950
`GarbageCollector`	Chi	5,590	0	5,590
`SimplerHashMap`	Replacement	4,310	0	4,310
`BluetilesNetwork`	Chi	4,240	0	4,240
`Nexus.NexusTPIFRx`	Machine Java	3,850	0	3,850
`ArrayList`	Java	3,670	0	3,670
`BluetilesMicroblazeRuntime`	Chi	3,660	0	3,660
`ProcessorDrv`	Machine Java	3,400	0	3,400
`Utilities`	Chi	3,330	0	3,330
`PriorityQueue`	Java	3,270	0	3,270
C and Java entry points	Chi	2,850	8	2,870
`SpeedTest`	Application	2,840	0	2,840
`CastingString`	Replacement	2,650	0	2,650
`ArrayDeque`	Java	2,590	0	2,590
`LinkedList`	Java	2,590	0	2,590
`Collections.SynchronizedMap`	Java	2,540	0	2,540
`AbstractMap`	Java	2,420	0	2,420
`Nexus`	Machine Java	2,360	0	2,360
Total for largest 20	all	84,000	28,900	113,000
Total	all	168,000	28,900	197,000

Table 5.3: *The 20 largest contributors to the overall static memory consumption of the* `SpeedTest` *microbenchmark. Rows in* `teletype` *font indicate compiled Java classes. In total there are 203 components that contribute to* `SpeedTest`*'s static memory consumption. The data in this table is from a build with the* ***Bs, -Os, min*** *set of characteristics described in figure 5.16*

ded contexts. The remaining eight largest contributors are runtime and framework components that originate from Machine Java and Network-Chi.

The `SimplerClass` class is Network-Chi's implementation of `java.lang.Class`. Although `SimplerClass` is the largest single contribution to the application's size from a Java class, much of its content is actually generated by Network-Chi at compile time. In particular the `SimplerClass` provides the garbage collector with information about where references are located in each object type. This table of reference offsets is the reason for `SimplerClass`es data memory consumption. The garbage collector itself is the single largest consumer of code memory from non-generated Java code.

When compiling for the Blueshell target (as is the case in table 5.3) the application will execute directly on the processor with no underlying operating system. Slightly more work is required in the runtime to support a bare-metal environment but there may also be differences in the relative efficiencies of each target's C compiler.

- The features enabled in Network-Chi during compilation. Network-Chi allows a number of Java features to be selectively enabled at compile-time. The features that have the most important effect on the code size (and runtime performance) of an application include:

  Virtual machine generated exceptions During execution of standard Java the JVM will instantiate and throw exceptions if the code performs an operation on a runtime value that is not appropriate. These exceptions include:

  - `NullPointerException` if a reference to **`null`** is used as an object (for field accesses, synchronisation or method invocation).
  - `ArrayIndexOutOfBoundsException` when an attempt is made to access an element beyond the valid bounds of an array.
  - `ClassCastException` when an object is assigned to a reference with an incompatible static type.
  - `ArrayStoreException` when an attempt is made to store an object in an array with an incompatible *runtime* member type. The static type of an array reference is insufficient to determine if an object can be stored in an array.

– `ArithmeticException` when a divide by zero is attempted.

Considerable savings in code size can be achieved by disabling support for one or more of these virtual machine exceptions, but disabling any of these causes a significant deviation from Java's expected semantics and will frustrate attempts to debug a faulty application. Applications which are written to depend on VM generated exceptions for their correct operation will no longer function correctly if these are disabled. The behaviour of many operations becomes target specific when VM generated exceptions are disabled. For example, when the feature for `NullPointerExceptions` is disabled any code that invokes a method on a **`null`** reference will apparently succeed if the method does not use its **`this`** reference, but if it does use **`this`** the outcome will depend on the target's response to a null-pointer dereference. On a POSIX operating system it is likely the process will crash with a segmentation violation or similar, but on an embedded processor such as the MicroBlaze the outcome can vary from triggering a hardware exception, to entirely undefined behaviour.

Ordinary, application allocated and thrown exceptions still operate as expected even when all VM generated exceptions are disabled.

Class names The ability for an application to obtain a `String` representing the name of an object's class at runtime is expensive as it requires a (sometimes long) string literal for every instantiable class in the application. Java's runtime representations of classes (the `Class<?>` class), which can be obtained via expressions on class literals (`Machine.class`) and objects (`someobject.getClass()`) may not even be used in an application but this feature is used by `Machine.log()` to record a human-readable description of the machine that invoked it. The ability to print the type of an exception is essential to debugging efforts and this is completely dependent on runtime support for class names.

This feature only strongly affects the read-only data of an application as it contributes to the used string literals. Disabling class name support has no functional consequences for most code. It is not expected that Java code will predicate its behaviour on the contents of the `String` returned by `getName()` on a `Class` object, Java's polymorphism and **`instanceof`** operator are much

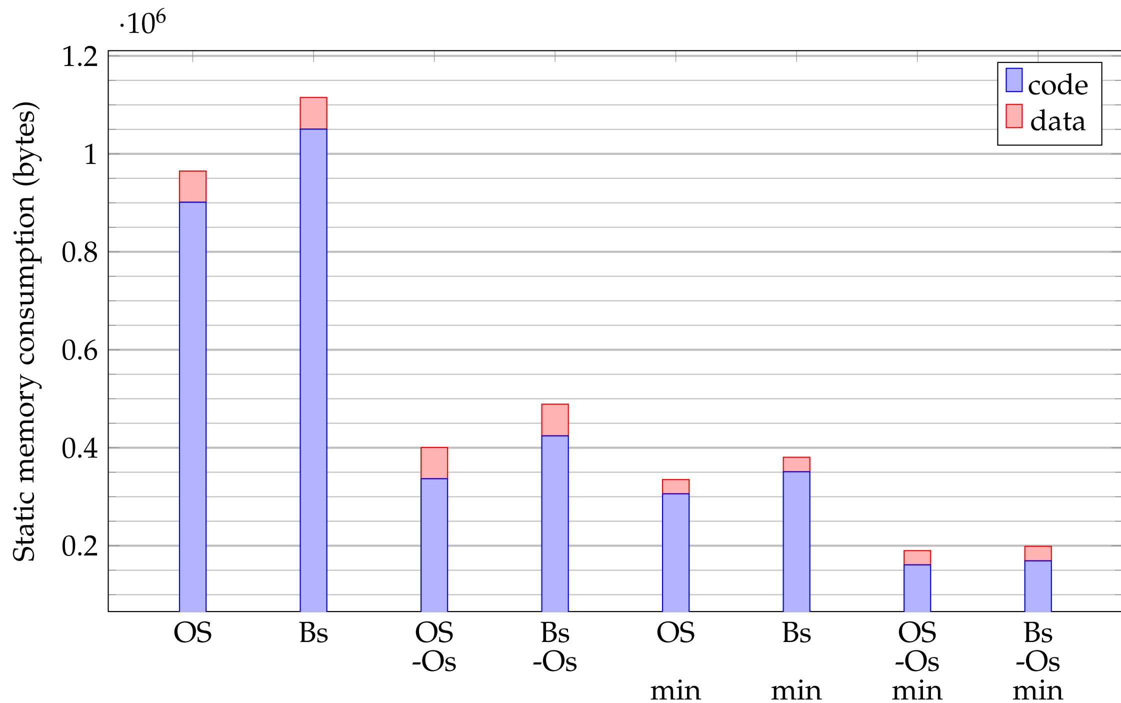
safer techniques for switching behaviour according to the runtime type of an object.

Stack trace support The ability to print a human-readable call-stack trace associated with an exception is indispensable for debugging code, but is also extremely expensive. Stack trace support implies a huge burden for both code and data memory consumption. Significant additional code is required for tracking, manipulating and printing call stacks. Additional data consumption is required as every class name, method name and filename containing the classes must be available for printing at runtime. As with class name support, sensible Java code does not predicate its functional behaviour on the contents of exception stack traces, so there are no functional consequences to disabling this feature.

- The optimisation level of the downstream C compiler used to build the target-specific binary. The popular GCC [67] family of compilers support several optimisation levels ranging from 'none' to aggressive (and often unsafe) optimisations [2, §3.10].[10] GCC's `-Os` (optimise for minimum code size) is the most useful for Network-Chi. Applications compiled with `-Os` benefit from considerably reduced code size but also improvements in execution performance. Execution performance is improved because size-optimisation includes all safe optimisations that do not typically also increase binary sizes, and because smaller code is more likely to fit into whatever processor cache is available.

  An application compiled via Network-Chi and GCC will also see an apparent increase in available *runtime* memory; code-size optimised applications can work on more objects before an `OutOfMemory` exception is thrown by the allocator. This is because the garbage collector implementation (see section 4.7.2.5) considers all objects referenced by the stack to be live, and Chi emits C-code that uses single-assignment for temporary variables. Even simple Java methods can have hundreds of temporary variables in the C implementation. When this single-assignment style of code is compiled without optimisations every method that is live on the call stack will continue to keep alive every object that it has ever referenced with

[10] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` provides an exhaustive list of supported optimisation options.



Build characteristics	
OS	build targeted a POSIX OS
Bs	build targeted the Blueshell example NoC
-Os	build used size optimisation
min	build excluded VM exceptions and stack trace support

Figure 5.16: *A plot of the `SpeedTest` application's static memory consumption when built with various combinations of options.*

a unique expression. However, when optimisations are enabled the C compiler will reuse stack memory and registers for values that are never used again, and this allows objects that have finished their lifecycle to become collected even if the method that allocated it has not yet returned.

The impact of these various compilation options on the `SpeedTest` microbenchmark's (see section 5.4.2.3) static memory consumption is illustrated in figure 5.16. It can be seen that the target architecture has a small impact on code size and no impact on data size. Each of the **-Os** and **min** build options have a dramatic impact on the application's code size, and can be combined to produce binaries with less than 20% of their original static memory requirements. As one might expect, the size optimisation (**-Os**) has no impact on data consumption but excluding non-essential Java features (**min**) substantially reduces compiled-in data.

Figures 5.17 and 5.18 plot code and data requirements for a selection of twelve Machine Java applications. The applications considered include:

Microbenchmarks discussed in section 5.4.2.3.

NoMachine This application never starts the Machine Java framework; `Start.start()` is never invoked.

AMachine Starts the framework and logs "Hello World!" in the start machine's `internal()` method. This application contains almost no code but creates a dependency on a substantial proportion of the Machine Java framework.

Eratosthenes Is a distributed sieve of Eratosthenes. The start machine feeds a stream of integers into the first machine of a pipeline. Each machine filters (discards) the integer if it is a multiple of the machine's filtration number. If the number is not filtered, its passed to the next machine in the pipeline for consideration or a new machine is created if there is not yet another pipeline stage. When a new machine is created this indicates that a new prime number has been discovered and the start machine is notified of the new prime. Machine Java's implementation on Network-Chi renders this construction embarrassingly inefficient. The communications overhead when sending integers between the pipeline stages dominates (by several orders of magnitude) the single division operation that each pipeline stage performs for each filtration. More sophisticated compilation strategies are required to enable efficient execution of such finely divided workloads. At runtime this application will have as many pipeline stages as prime numbers that have been discovered.

Dining Philosophers The `Dining` and `PentagonalTable` applications are the dynamic and static dining philosopher implementations, respectively. The dynamic implementation is reproduced in appendix C.4.

FlowController is the water tank level control example from chapter 3. This application is reproduced in appendix B.

Collatz The `CollatzBB` and `CollatzRPC` applications generate a sequence of integers[11] related to the Collatz problem [113], also known as the $3n + 1$ problem. The Collatz

[11] This is sequence number A006877 in the *Online Encyclopaedia of Integer Sequences* [1].

conjecture states that for the following function:

$$n'(n) = \begin{cases} 3n + 1 & \text{if } n \text{ is odd} \\ \frac{n}{2} & \text{if } n \text{ is even} \end{cases}$$

n' will always eventually reach 1 for any natural number input with repeated applications of the function. The example system calculates the sequence of starting numbers such that the count of applications of the function needed to reach 1 is greater than any previous number in the sequence generated. These are the referred to as the *high-water* marks. These applications provide identical runtime behaviour but the `CollatzBB` implementation only uses **BoundedBuffer** channels to communicate between the worker and dispatcher machines, whereas the `CollatzRPC` uses **RemoteProcedureCall** channels.

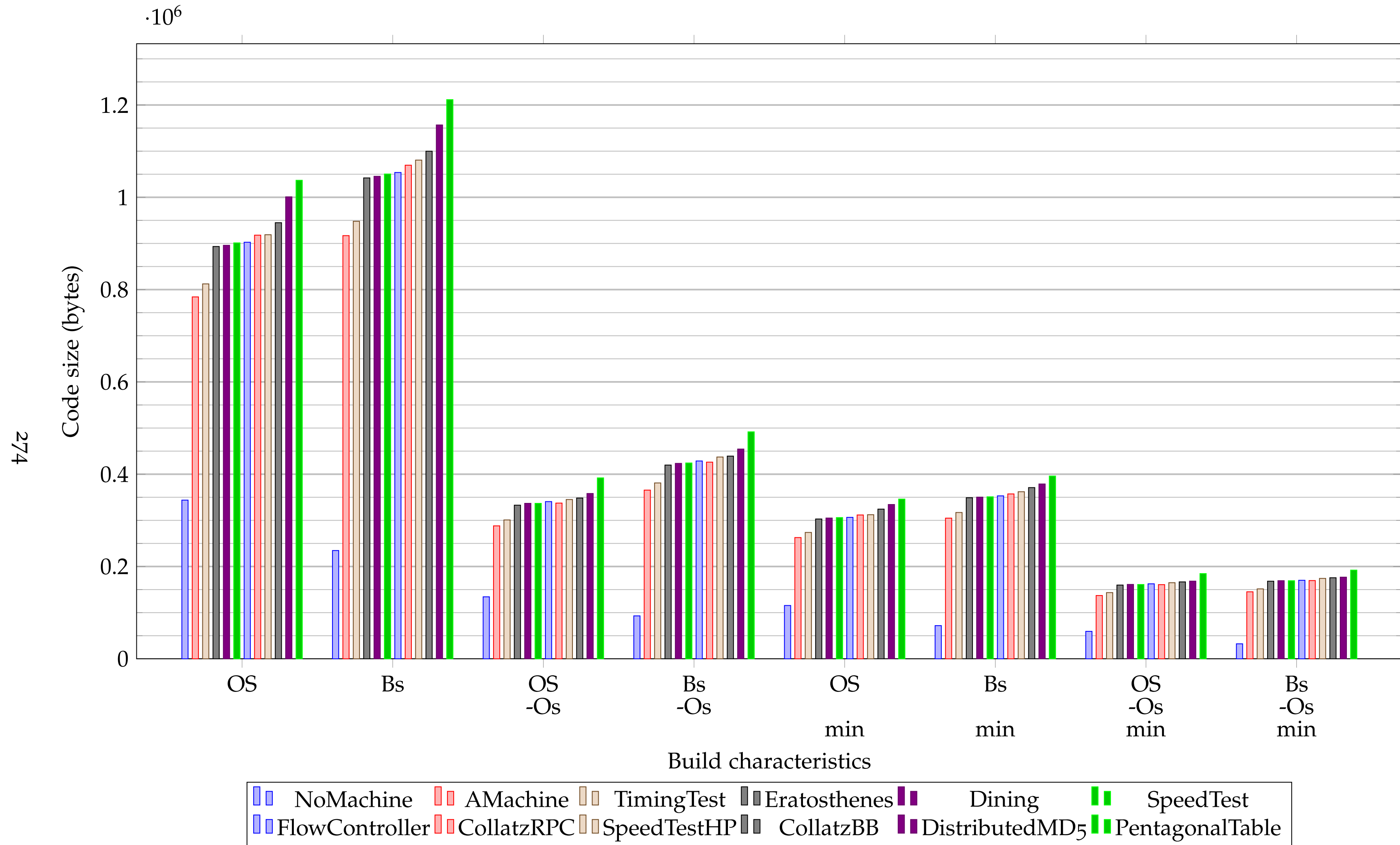


Figure 5.17: *A survey of code sizes for a suite of twelve Machine Java applications. See figure 5.16 for a description of the build characteristics.*

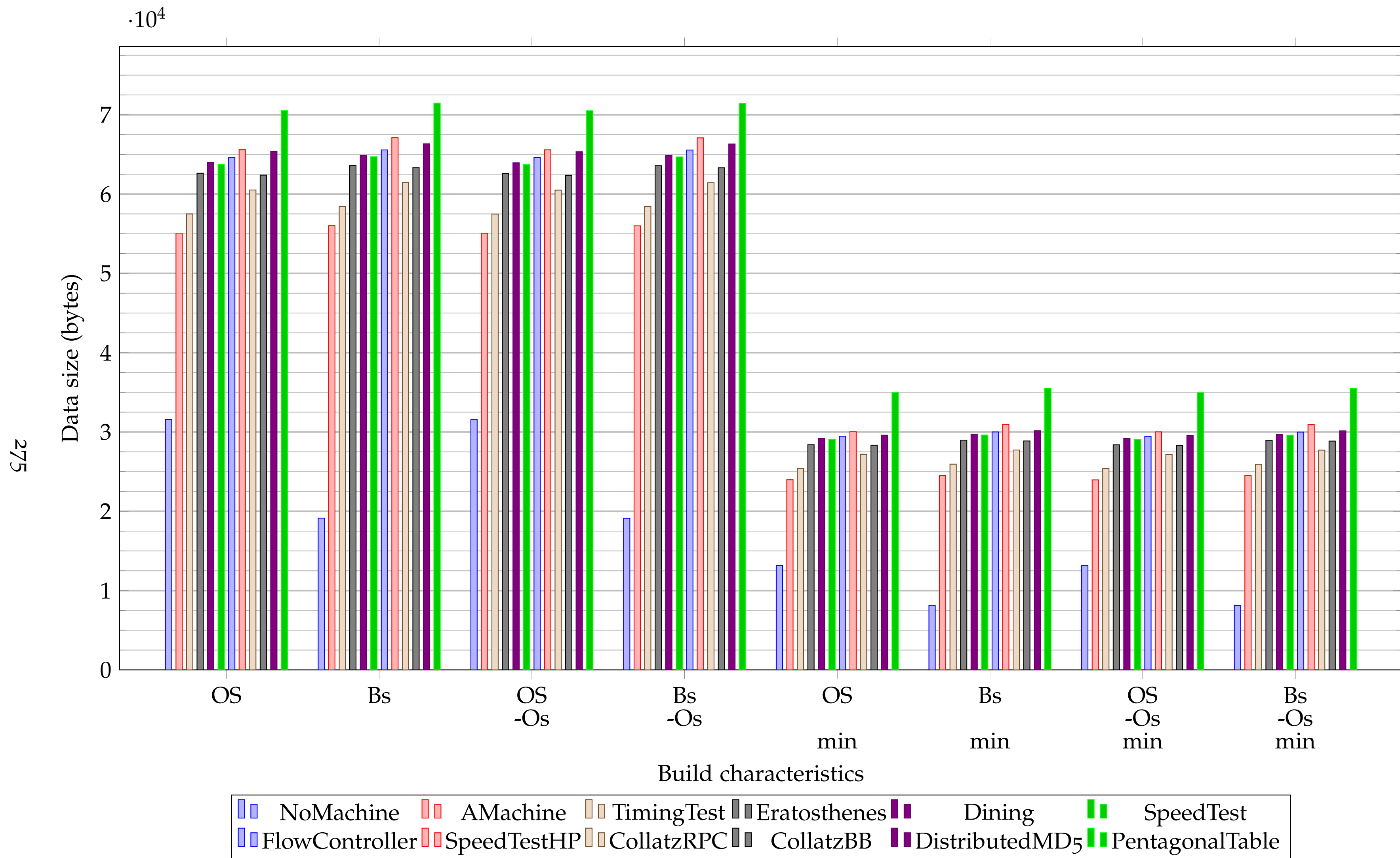


Figure 5.18: *A survey of data sizes for a suite of twelve Machine Java applications. See figure 5.16 for a description of the build characteristics.*

A number of observations can be made from the data presented in 5.17 and 5.18:

- When comparing the size of `NoMachine` on the POSIX target and the Blueshell target, the Blueshell target requires less code. However, all other applications consume more code on the Blueshell target. This can be explained by the Blueshell target having a *smaller* Network-Chi runtime implementation but a less efficient instruction set or C compiler. The savings achieved by the smaller runtime are absorbed by the ISA/compiler inefficiencies before even the next smallest application is considered. This is a somewhat surprising result though, as the OS target would be expected to benefit from an OS that can assist with networking, timing and memory management. On the Blueshell target the Network-Chi runtime has to manage all of these itself.
- By comparing the sizes of the `NoMachine` and `AMachine` applications, it can be seen that the Network-Chi runtime accounts for approximately one third of the framework's code overheads, with Machine Java responsible for the remaining two thirds.
- The build characteristics **-Os** and **min** reduce memory consumption by a similar factor regardless of the application.
- The Network-Chi and Machine Java runtimes are a far more significant contributor to static memory consumption than the applications. Vastly more complex applications would be required before the application is responsible for the most code.

5.4.2 Experimental Methodology

The experimental results presented in this thesis are the result of a considerable infrastructure. Machine Java and the Chi compiler combined amount to nearly 50,000 lines of Java and there are several thousand lines of supporting infrastructure to dispatch experimental trials to a target, gather the results and then convert the logs into a format that can be analysed and typeset. Even with a copy of the original Machine Java source code, it would be a considerable undertaking to duplicate this experimental environment due to the large number of complex dependencies. The purpose of this section is to describe the experimental methodology such that a third party may attempt to reproduce the experiments in future.

5.4.2.1 Reliable Experiments on the Blueshell Platform

The intent of the experiments in this chapter is to explore the behaviour of the machine-oriented programming model on an example of a non-cache-coherent (*communication-centric*) multiprocessor hardware platform. As such, the experimental methodology follows the requirements of the hardware platform described in section 5.2.

An important characteristic of the evaluation platform is that it behaves completely deterministically: given identical starting conditions an application will always execute identically, down to each clock cycle on every router and processor. None of the example applications or microbenchmarks accept external input during operation. The applications do output data to an external serial line during execution but this is not externally synchronised; there is no hardware flow control. The importance of this is that the external environment has no effect on the execution trace of the platform. No difference in execution has ever been observed between two runs of the same *deterministic* program code in the hardware platform. Non-deterministic code, such as code that uses psuedo-random numbers will not necessarily execute identically each time as the random number generator is seeded based on the number of clock cycles since the FPGA was configured, and this number of cycles will not always be identical as the time between FPGA configuration and bootloading completing is variable.[12] A future reproduction of these or similar experiments could guarantee repeatability even for applications that use pseudo-random numbers by either of:

- Clearing the cycle-counter hardware in all processors simultaneously after bootloading has completed, but this may be expensive to implement in hardware.

- Fixing the random number seed in Java's `java.util.Random` class to a value not dependent on the system timer.

- Waiting for a predefined time after FPGA configuration to release the bootloaded program code. As long as this predefined time is comfortably longer than the worst case bootloading time then the program code will always execute at a fixed offset from FPGA configuration and therefore the random number generator will always return the same values.

[12] This is a consequence of the variable timing inherent in the python host application running on a standard PC.

5.4.2.2 Platform Configuration

The hardware platform configuration has been described in section 5.2.2, and experiments on the OS target were performed on a standard PC architecture machine with the following configuration:

Hardware 4×Intel Xeon CPU E5-4607 v2 @ 2.60GHz. Each has 6 cores and 12 hardware threads per processor, resulting in 48 hardware threads in total. 128GiB RAM.

Operating System Debian GNU/Linux 7 (wheezy) using Linux 3.2.63-2+deb7u1 x86_-64.

C Compiler GCC version 4.7.2 (Debian 4.7.2-5)

The OS target is used to provide an indication of performance on a complex, modern processor architecture and is not intended to be a primary evaluation platform for the merits of Machine Java or the MAA in general. For this reason, no attempt has been made to control for the noise introduced into the results on the OS target due to interference from other system processes and concurrent access by other faculty members.

In the remainder of this section the results presented are for code compiled with size minimising compilation options:[13] `-Os` for the downstream C compiler and VM exceptions and stack trace support disabled in Network-Chi. In all cases Java version 1.7.0_79 (OpenJDK[14] [151]) was used as Network-Chi's source of Java libraries. Both the Blueshell and OS targets use 48KiB Java heaps at runtime, but the OS target does not share the Blueshell target's 16KiB stack constraint.

5.4.2.3 Microbenchmarks and the Experimental Procedure

To determine the overheads implied by Machine Java and compilation via Network-Chi, a number of *microbenchmarks* have been constructed. In a sense the applications described in this section are *real* benchmarks as they are complete applications, however their scope is highly constrained in order eliminate as many confounding effects as possible from the results. The results of the work done by these applications (their output) has no real world meaning; only measurements of the operation of the applications are used.

[13] Class names remain enabled as they have an insignificant impact on *code* size and execution speed.
[14] OpenJDK Runtime Environment (IcedTea 2.5.6).

Two separate microbenchmarks are used to investigate the communications and computation implications of Machine Java, each with several variants to enable the investigation of more specific issues. Communications overheads and scaling are investigates with the *SpeedTest* series of microbenchmarks which are discussed in section 5.4.2.4. Computation overheads and scaling are investigated with the *DistributedMD5* series of microbenchmarks, and these are discussed in section 5.4.2.4.

These microbenchmarks all follow the same basic procedure to gather results:

1. On system startup (in the microbenchmark's `Start` class's `internal()` method), a machine parseable header is emitted identifying the benchmark's name and any relevant experimental parameters (such as the message size, or batch size).

2. The `Start` class periodically emits machine parseable log entries of the amount of 'work' that the application has completed by receiving messages from worker machines that it has also created. Only the `Start` class logs messages to avoid introducing interference between machines as they compete for the logging resource. These log entries also include other relevant parameters such as the number of active worker machines.

3. The `Start` class also periodically creates new worker machines. This worker-addition interval is much larger than the time required to complete a unit of work, and also much larger than the interval between progress reports from workers back to the `Start` machine.

Within a particular experiment variability in the performance of a metric such as computations per second, or bytes transferred per second, is accounted for by allowing enough time for a large number of these events to have occurred and then computing the arithmetic mean of the measurements. It is a limitation of these experiments that the precise distribution of event characteristics is not captured due to limitations of the hardware platform: insufficient memory is available to buffer data samples and the communications speed of the logging interface is too low to continuously transfer data samples.

Variability of the experiment as a whole is not accounted for as the experiments run identically every time. There is no requirement for repeated execution of experiments on the hardware platform, the variance is known in advance to be zero. It is a very different situation on the noisy OS target which has very noticeable variance in results between experimental executions. However, only the very coarse and general shape of

the results are considered from the OS target as this is neither a resource constrained nor a communication centric architecture. Experiments were not repeated on the OS target as this thesis does not require confidence at fine granularity on a platform outside of the intended domain.

5.4.2.4 Execution of Experiments

For execution of experiments a basic test-runner framework was constructed. This enabled a large number of experiments to be executed and results to be recorded. For the performance experiments (the ones in the remainder of this chapter) the procedure is as follows:

1. The root experimental script[15] is invoked with the name of the experiment to execute and the target platform supplied as arguments. The name of the experiment is the Java class to be used as the entry point to the application, such as `examples.speedtest.SpeedTest` (see appendix C.1).

2. The script cleans up the working environment of all temporary compilation artefacts from previous experiments that could interfere with the build or execution of this experiment. This also includes termination of previous stale experiments if they are still consuming hardware resources.

3. The script now executes the application once for each of the twenty processor-counts to be tested:

   (a) A new log file is created and the parameters of the experiments are emitted into the log, including: the application to be compiled, the options supplied to Chi, the target platform and any experimental parameters supplied.

   (b) Chi is now invoked with the necessary parameters and its output is captured into the log. Chi is also instructed to execute the binary it has built once compilation has finished. All output from the execution of the binary is also captured into the log. For the OS target execution of the binary is simple: the resulting binary is executed on the same hardware. For the Blueshell hardware it is more complex:

[15] These were implemented as Bash shell scripts.

i. Chi invokes a helper script to execute the built binary. This helper script first prepares the binary for execution on the microblaze processors in the Blueshell hardware. This requires conversion of the elf-format executable into ihex format and then expansion of the ihex representation into a flat binary that matches the memory layout of the hardware.

ii. The helper script then begins the process of programming an available FPGA board with the Blueshell NoC instance and waits for the NoC hardware to become available. Once the hardware is operational, the application binary is streamed into the off-chip DDR memory. When this process has finished the *first* processor of the platform is sent the command to branch to the boot vector of the binary. The Chi binary now handles the startup of the processor from this point, and subsequently the Bluetiles Machine Java runtime will boot other processors if configured to do so.

(c) The application now executes and periodically logs useful information.

(d) When the application has completed, an error condition has been detected or a timeout in the experimental execution script has happened, then the experiment is finished and the process can be terminated safely. At this stage the Blueshell hardware platform is deprogrammed so that the FPGA board can be reused for future experiments. There is no possibility of interference between experiments on the hardware platform as the hardware is fully cleared and reconfigured for every experiment, this ensures the processors are in the same clean state each time.

4. The next iteration begins until there are no more platform configurations to test.

5. The captured logs (an abridged example of which can be found in appendix D) can now be processed into the plots which can be seen in the remainder of this chapter.

The SpeedTest Microbenchmark

The *SpeedTest* microbenchmark is a simple application that attempts to measure the communications throughput achieved between a number of communicating pairs of

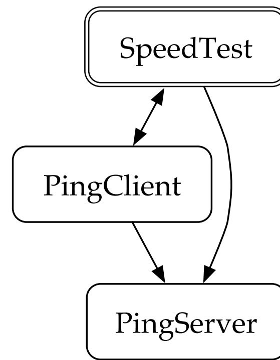


Figure 5.19: *The SpeedTest microbenchmark's static machine dependency graph.*

machines. The machine dependency graph can be seen in figure 5.19. The base version of this microbenchmark is reproduced in full in appendix C.1.

At runtime the SpeedTest application has a number of communicating pairs of `PingClient` and `PingServer` machines. A `PingServer` machine has a single **RemoteProcedureCall** channel (`replyService`) which replies to all queries with the data sent[16]. Each `PingClient` machine sends a message of a fixed size to its paired `PingServer` machine. As soon as a `PingClient` receives a reply it issues the next query. Periodically each `PingClient` sends a message to the application's `SpeedTest` machine with an indication of the number of exchanges that were performed in the interval and how long the machine had to wait in total for its replies.

In this application only the start machine (`SpeedTest`) logs any messages, avoiding contention for the platform's message logging device. After the start machine has been created the following sequence of events occurs:

1. A periodic event source is created to create a new pair of machines on a regular schedule (every ten seconds).

2. A delayed event source is created to terminate the experiment after a fixed interval (ten minutes).

3. The benchmark identifies itself and creates the first pair of communicating machines.

Before each new pair of machines is requested, the application logs the aggregate data throughput, latency and number of messages exchanged. The message count and data

[16]This is similar in spirit to the ICMP Echo/Reply [170] protocol mandated of internet hosts by RFC1122 [34].

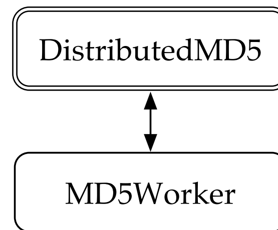


Figure 5.20: *The DistributedMD5 microbenchmark's static machine dependency graph.*

throughput figures are the sum across all communicating pairs. The recorded latency is the average round-trip time for a message, calculated by summing all time between when `PingClient` sends its request and when the same `PingClient` machine begins to handle the event for receiving a reply. This total latency measured in a fixed interval is then divided by the number of message exchanges in the same interval, yielding an arithmetic mean of the latency per exchange.

The Machine Java runtime supports a *fast-fail* mode for experimental data collection. When in this mode if an uncaught exception propagates out of any event handler the runtime will attempt to terminate the whole application. Failures of application logic do not occur during execution but memory exhaustion (`OutOfMemoryError` exceptions) are common when a large number of machines are allocated to few processors.

The DistributedMD5 Microbenchmark

The *DistributedMD5* and SpeedTest microbenchmarks share a similar general pattern in operation, but the DistributedMD5 application aims to measure the total computational throughput of the application rather than a communications throughput. DistributedMD5's machine dependency graph is shown in figure 5.20, and is reproduced in full in appendix C.3.

At runtime the start machine (`DistributedMD5`) periodically requests additional worker machines (`MD5Worker`). Each worker machine repeatedly computes the MD5 [183] cryptographic message digest (a *hash*) of an incrementing 64-byte buffer. After each new hash is computed the `MD5Worker` machine uses a `Yield` to allow other event handlers a chance to execute. In most cases there are no other pending events and the worker will begin another computation. A third-party all-Java implementation of the MD5 algorithm ([153]) is used as the standard Java libraries for message digests (`java.security.MessageDigest`) use non-Java language implementations and

are therefore incompatible with the Chi compilation workflow.

Each `MD5Worker` periodically reports how many digests it has computed to the `DistributedMD5` machine, which are logged by `DistributedMD5` in the same way as `SpeedTest`. The MD5 algorithm is a sizeable computational kernel, requiring 6,480 bytes of code memory on a Blueshell processor.[17] This makes it the single largest code contributor in the DistributedMD5 application, considerably larger than the 5.6KB required by the garbage collector. In contrast the Java code of the `MD5Worker` machine itself (a machine and two event handlers) only requires 1,704 bytes for its three classes.

5.4.3 Communications Overheads

In Machine Java the serialised version of an object communicated between a pair of machines is determined either by the object itself (see section 4.7.3.4) or by Network-Chi's utilities for flattening and inflating common Java types. In either case it is possible to determine at design and compile-time what the overall serialised format for an object will be. The size of an object's serialised format and knowledge of the wrapper objects used by Machine Java during communication can provide a coarse indication of the static communications overheads. However there are a number of factors that hinder an *a priori* estimation of communications overheads:

- The application can be executed on different platforms with unknown communications characteristics and inefficiencies.

- Interference between unrelated transactions across the platform's communications media may can be unpredictable, especially as the runtime loading of an application cannot always be known ahead of time.

- The Machine Java implementation has substantial freedom to chose its underlying communications protocols, with potentially significant non-functional consequences.

For example, when a `PingClient` issues a request to a `PingServer`'s **RemoteProcedureCall** channel it sends a `PingMsg` object. This object implements the `Flattenable` interface. For a five-byte `PingMsg`, ten bytes are serialised: a one byte marker to indi-

[17] Compiled with the same options as the SpeedTest application components shown in table 5.3.

cate the serialised message represents a `Flattenable` object, four bytes for the integer identification of the `PingMsg` class, and finally the five bytes for the `PingMsg`'s payload.

In turn this `PingMsg` is wrapped in a `BidirectionalQuery` object by the **RemoteProcedureCall** connector (which provides return path information), and this is then wrapped in a `NexusDatum` object records sufficient details of the destination channel to allow the remote processor to understand the message. Overall the five-byte message becomes 76 bytes of serialised Java objects. A breakdown of this overhead may be seen in figure 5.21. In figure 5.21 serialisation starts at the outermost `NexusDatum` object and proceeds as defined by the class's `.flatten()` method. In this example the `.flatten()` methods just serialises each field in order.

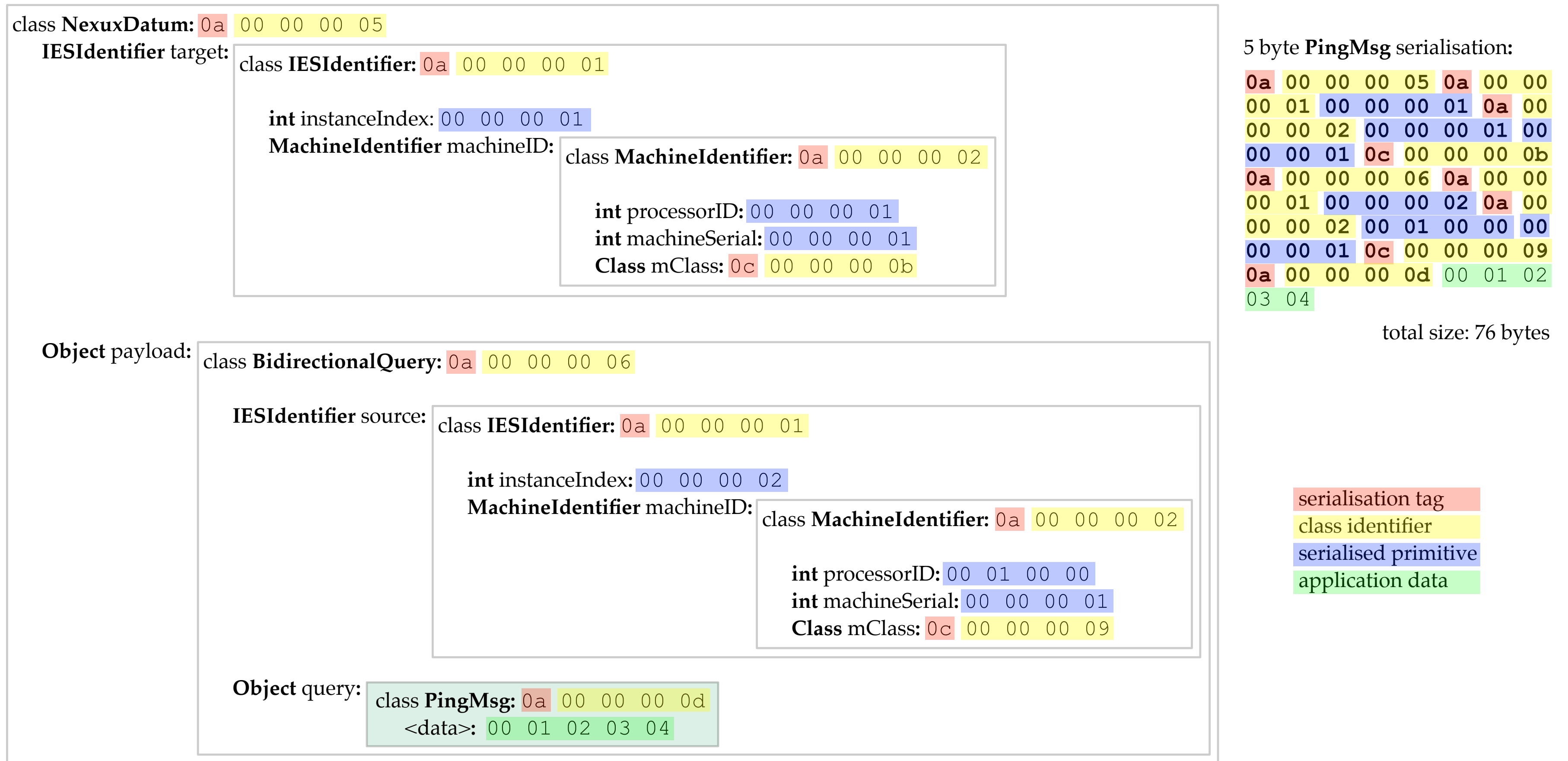


Figure 5.21: *A 5 byte* `PingMsg` *object becomes 76 bytes by the time it reaches the network driver. The wrapper objects add a considerable overhead. Tag* `0x0a` *indicates the next item in the stream is a flattenable object (see section 4.7.3.4), and tag* `0x0c` *indicates the next item is a class identifier.*

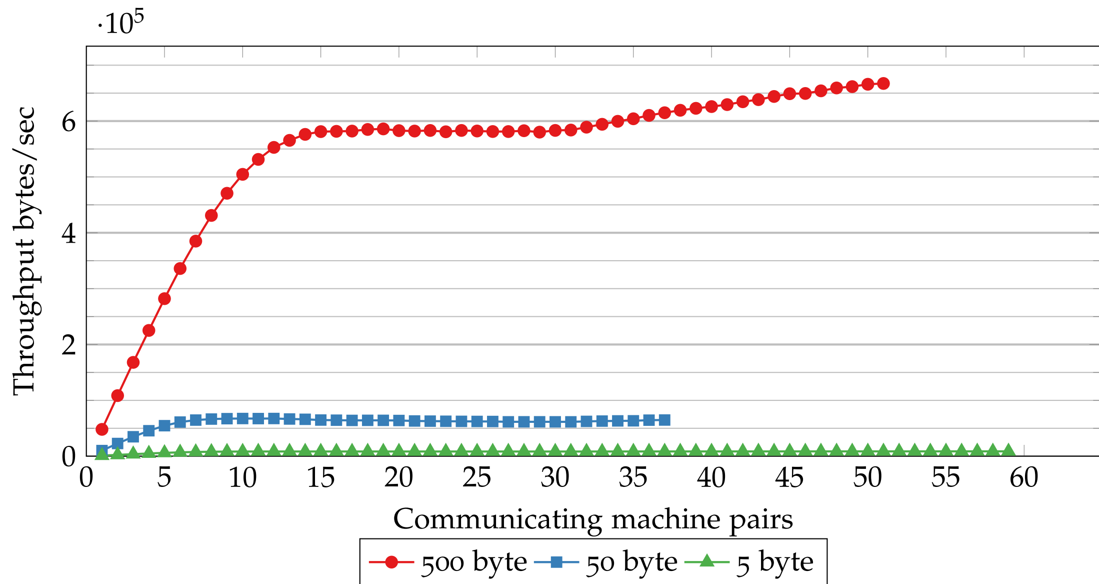


Figure 5.22: *A comparison of the total useful throughput achieved between all machines as the machine count is increased, and the message size is varied. These results are for the Blueshell example platform with use size-minimising compilation options. Three different message sizes are considered: 5, 50 and 500 bytes.*

The inefficiency introduced by wrapping a communicated object *at least* twice is naturally dependent on the size of the message. The lower levels of the communications stack will also introduce their own headers and inefficiencies. Blueshell prefixes network messages with an eight byte header and UDP used by the OS targets requires a 28 byte-per-message overhead. Figure 5.21 highlights sources of inefficiency in the message serialisation: Each of the 'serialisation tags' (see 4.7.3.4) are relatively small at only a byte, but there are nine of them in the serialised message. There are also nine class identifiers that are embedded in the message, all of which use a full 32 bit integer. Note that in figure 5.21 all of the class identifiers are very small integers as this example packet was serialised in a JVM and the JVM implementation only allocates integral class identifiers on demand. During compilation with Chi all classes are assigned an identifier ahead of time and will not necessarily have such low values.

Figure 5.22 plots how the measured throughput of a SpeedTest application varies according to the number of machines and the message size in use. For this experiment the full 32-processors of the Blueshell example platform were used, and the same 8×4 platform was used for an execution on the OS target, shown in figure 5.23.

A number of observations can be made about these results:

- Five-byte messages are confirmed to be extraordinarily inefficient.

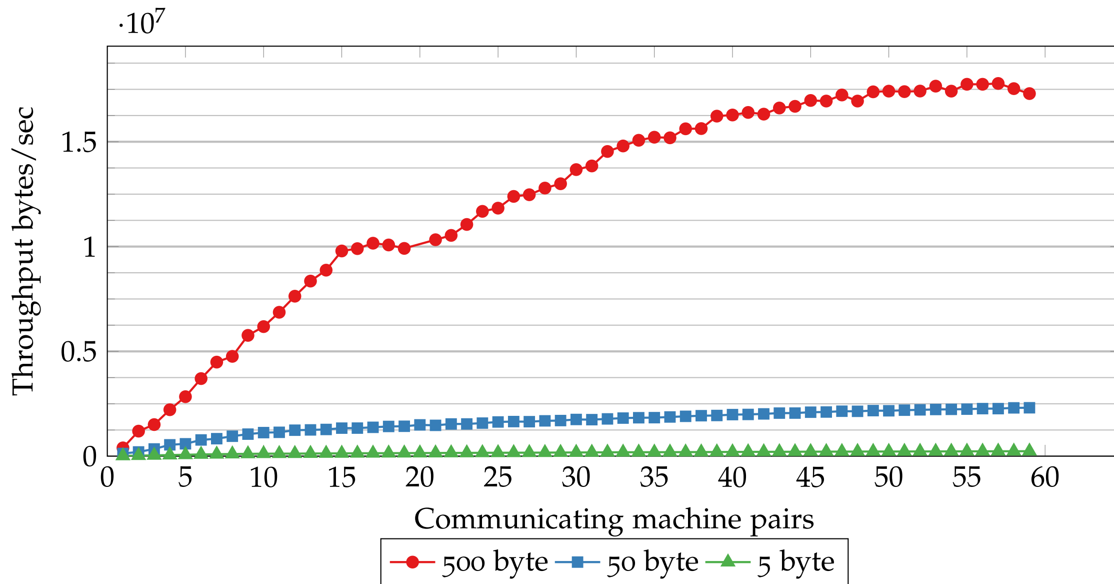


Figure 5.23: *The SpeedTest message size experiment (see figure 5.22) executing on the OS target via Network-Chi. An anomalous result for 20 machines with a 500 byte message size, caused by interference on the testing platform has been omitted.*

- The Blueshell plots (figure 5.22), and particularly the plot for 500-byte messages appear to exhibit a linear region followed by a constant region and then another shallow linear region. The OS target experiment plots also appear to follow a similar pattern although it is much less clear. But in both cases, the plot for 500-byte messages has a tripartite shape.

  The length of the linear region at the start of each plot appears similar on both platforms. This hints at an architecture independent cause for the shape of the plots. As both the Blueshell and OS targets share the same platform representation in Machine Java, the same runtime mapping of machine request to processors will occur. Machine overmapping (>1 active machine per processor) may appear to explain the performance degradation after 16 machine pairs when a 500-byte message is used, but the linear region appears shorter for smaller message sizes. The sharp 'knee' in the OS plot for 500-byte messages at 15 machine pairs contrasts with the gradual reduction in throughput improvement seen on the Blueshell architecture between 11 and 15 pairs. This may indicate another, *architecture-dependent* effect is also present.

- The Blueshell plots are very smooth in comparison to the OS target plots. This is reasonable considering the Blueshell platform has no operating system and is

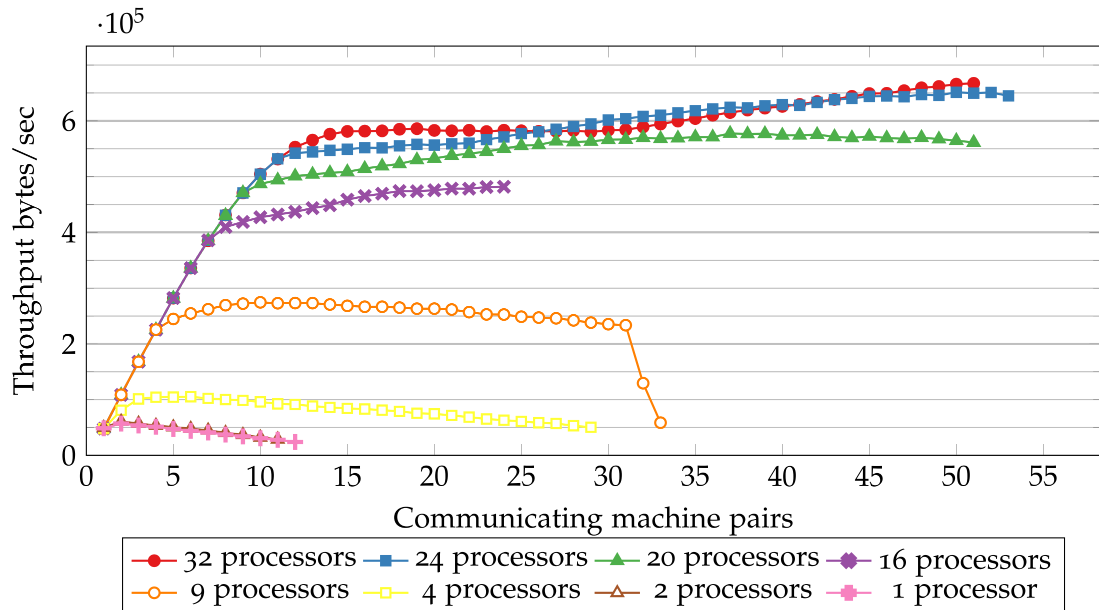


Figure 5.24: *A comparison of the total message throughput as the number of processors used on the Blueshell target is varied. Unused processors were not booted by the Network-Chi runtime so do not consume any memory network bandwidth.*

executing no other code except the application and the Machine Java runtime. Interference from other operating system responsibilities easily explains the OS target's noisier plots.

- The Blueshell network hardware is not perfectly reliable and fails unpredictably under heavy communications loading.
- Throughput is substantially lower than the maximum possible for either target architecture. This is easily explained by the inefficient object serialisation scheme chosen by Network-Chi. All data to be sent in an application has to pass through the processor on both send and receive sides. Hardware acceleration such as DMAs are not used to send Java objects.

This experiment raises a number of further questions:

1. Is the linear region at the start of the plots terminated by 'machine saturation' or is this a coincidence caused by the message sizes?
2. What is the likely cause of the tripartite plots for 500-byte messages?

To begin to answer the first of these new questions, the SpeedTest application can be executed on a variety of different network sizes. Given that the evaluation platform is

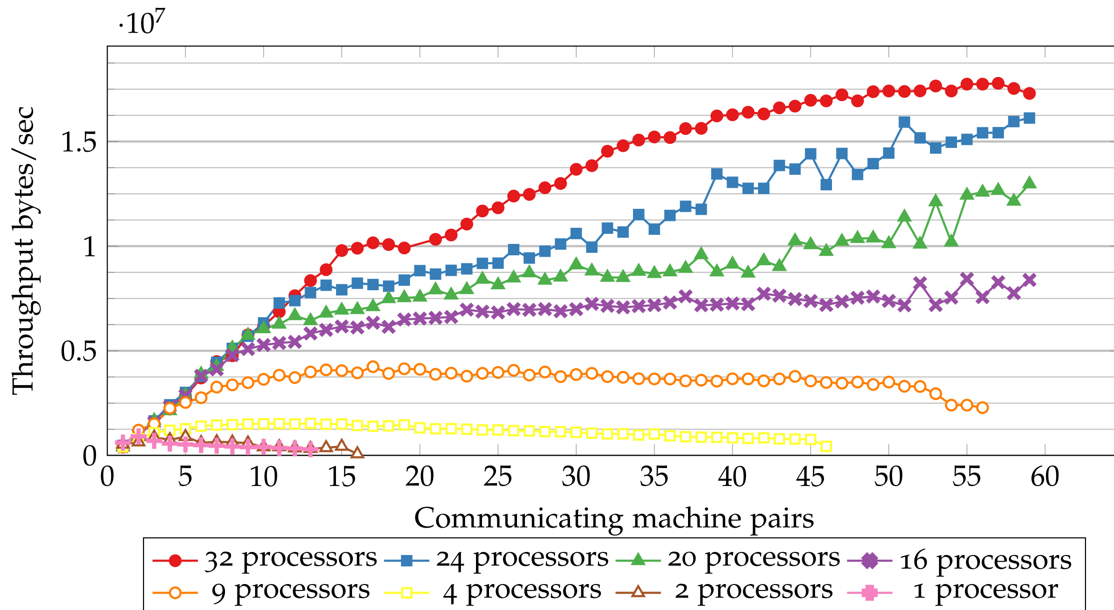


Figure 5.25: *The experiment shown in figure 5.24 executing on the OS target.*

an 8×4 mesh, there are 20 possible *rectangular* networks with unique processor counts that can be contained by it. This means an 8×4 hardware platform can emulate a multiprocessor system with any of the following processor counts:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 28 and 32 processors.

The SpeedTest application was executed again on a smaller subset of the hardware with each of the above processor counts. The throughput plots for eight of the processor counts are shown in figure 5.24 and figure 5.25 for the Blueshell and OS targets respectively. This experiment enables the following observations:

- The communications 'linear region', where communications throughput grows approximately linearly with respect to the number of communicating pairs, *does* appear to be related to the processor count in the hardware.

- On both targets each plot appears to approximately follow the same linear growth until the number of machines exceeds the number of processors. On Blueshell this divergence marks the end of substantial gains in throughput, indicating that performance is likely limited by processing power. On the OS target throughput continues to grow but more gently on all but the smallest of multiprocessors.

- For low processor counts (four and fewer) the processors do not have enough memory between them to host 60 machines, and `OutOfMemoryError` exceptions

halt the experiments before their conclusion.

The number of processors used in the platform appears to have some impact on the shape of the communications performance plots, with the highest processor-count experiments presenting the most clearly tripartite plots; the processor count alone does not explain the cause of this phenomena. However, the precise runtime allocation of machines to processors appears to offer a plausible explanation.

Machine Java's `XYNetworkPlatform` implementation (see section 4.5.5) does not make use of any application-specific information when suggesting processors for each machine request. It makes the assumption that all processors have access to all machine's code and exist at some (x, y) coordinates in a rectangular mesh. With little platform-specific information and no application-specific information to guide machine allocation, the `XYNetworkPlatform.getProcessorsForMachine()` returns an `Iterator` that will eventually list *all* processors in the platform.

For the results shown previously in this section the `XYNetworkPlatform` used a processor iterator that aims to be *position-relative* within the network. This processor iterator returns processors in approximate distance order from the current processor by ordering processors according to the rectangular 'shell' that they occupy around the current processor: the current processor is shell zero, the eight surrounding processors are in shell one, the next shell has sixteen processors (a 5×5 square with the inner 3×3 excluded), and so on. The intent is to suggest that the `ProcessorManager` pick local processors preferentially, but this is far from an ideal allocation scheme both in theory and practice. Figure 5.26 shows the allocation of the `SpeedTest` (ST) machine on processor $0, 0$ and the first 30 communicating machine pairs. The arrows indicate which `PingServer` (S) a particular `PingClient` (C) machine communicates with. The allocation order for machines is $(1, 0), (0, 1), (1, 1), (2, 0), (2, 1), (0, 2)$ etc.

A fixed, *sequential* processor iterator that always returns the processors in left-to-right, top-to-bottom (considering the layout of the processors shown in figures 5.26 and 5.27) regardless of the processor it is executing on, was used for another series of SpeedTest experiments. The allocation of the first 30 machine pairs to processors is shown in figure 5.27. This scheme is probably unacceptable if the network is thousands of processors wide but for this experiment it provides regular and quite short paths between communicating machines overall.

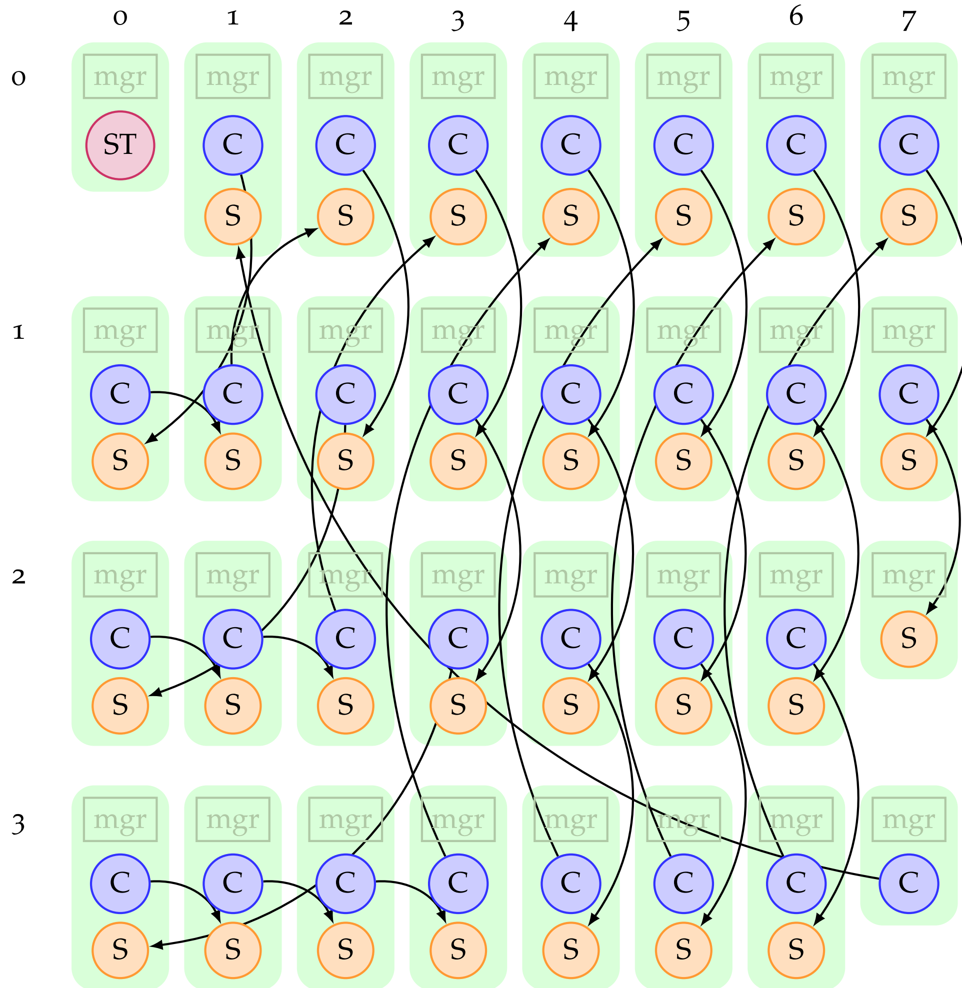


Figure 5.26: *The allocation of 30 pairs of machines in the SpeedTest application using the 'position-relative' processor iterator. Arrows indicate a reference from a* `PingClient` *(C) machine to a* `PingServer` *(S) machine. The green boxes indicate each processor in the platform, showing each machine that the processor contains.*

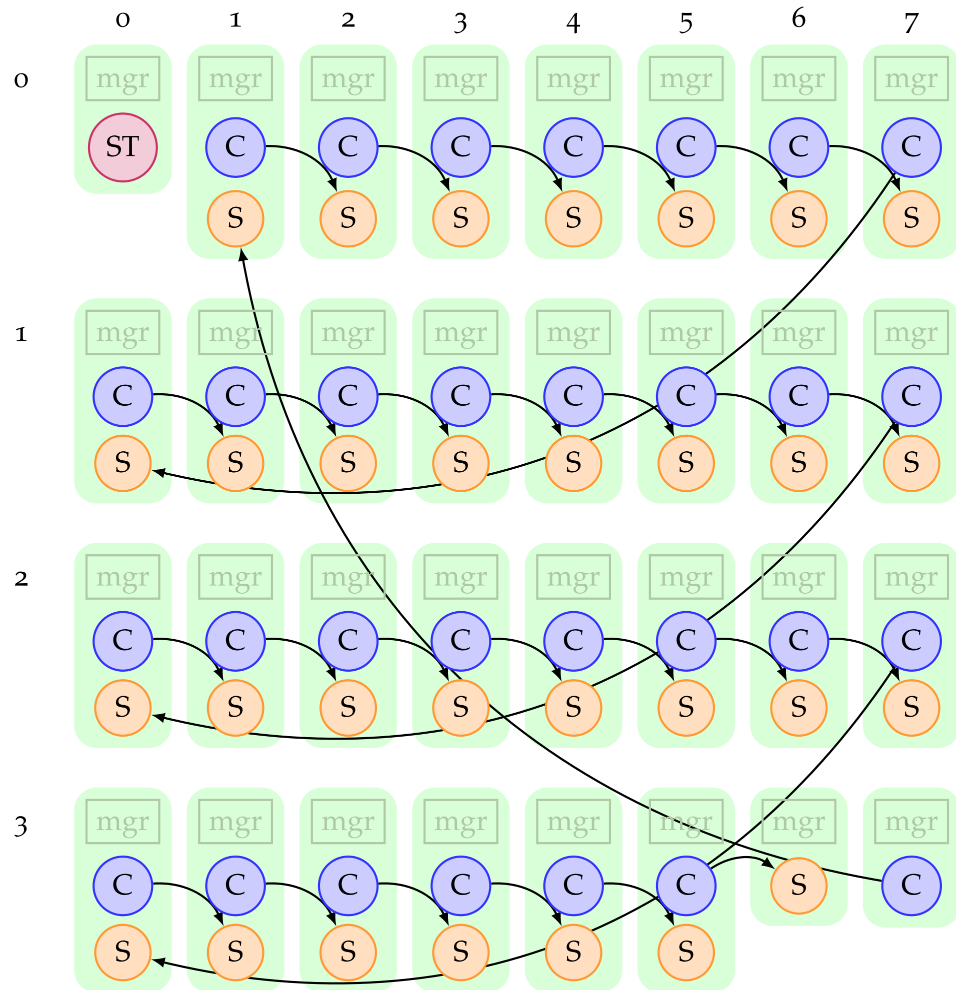


Figure 5.27: *The same application as shown in figure 5.26 but machines are allocated according to the 'sequential' processor iterator.*

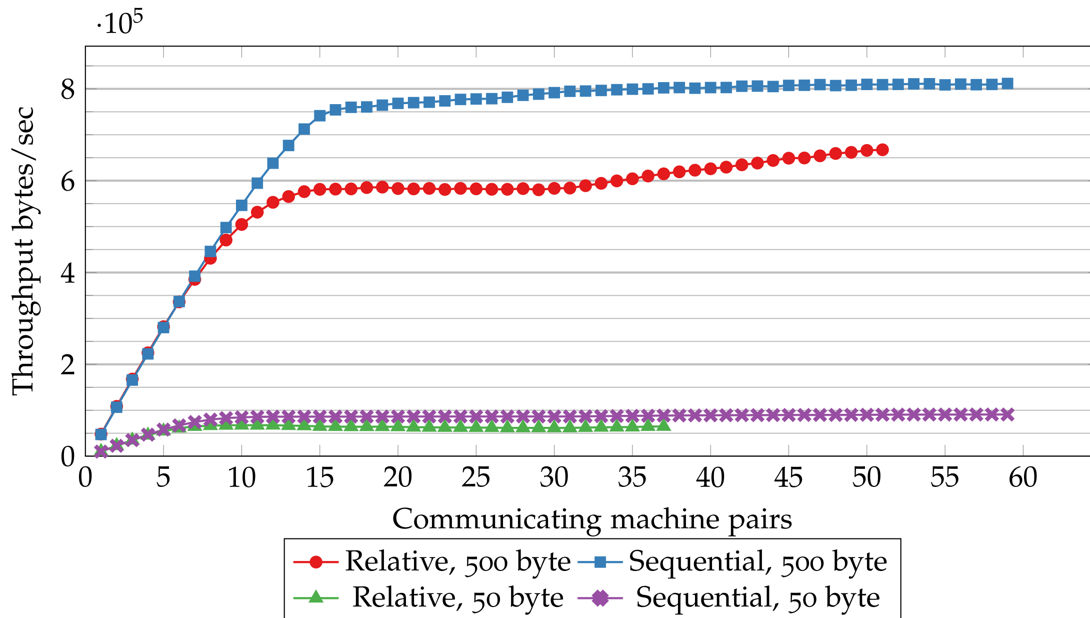


Figure 5.28: *A comparison of message throughput on the 32-processor Blueshell platform when both message size and machine iteration algorithm are changed. Representative machine-processor mappings can be seen in figures 5.26 and 5.27 for the 'relative' and 'sequential' algorithms, respectively.*

A comparison of the communications performance achieved with each of these processor iterators is provided in figure 5.28. It can be seen that the 'naive' sequential iterator outperforms the position-relative iterator for both 50-byte and 500-byte message sizes, and does not exhibit tripartite shape for the 500-byte message size. The reduced performance with the position-relative processor iterator on Blueshell may be caused by the additional network due to the (on-average) greater distance between the allocated machines.

A surface plot of 500-byte message throughput across all 20 processor counts can be seen in figure 5.29. The missing regions in figure 5.29 are where the number of machines exceeds the memory capacity of the platform.

Effects of Channel Protocol on Throughput

The SpeedTest application used in previous experiments (reproduced in appendix C.1) uses **RemoteProcedureCall** channels to communicate between the `PingClient` and `PingServer` machines. This pattern is intuitive for application design but does imply greater static communications overheads as each sent query is wrapped in a `BidirectionalQuery` object by the `RemoteProcedureCallConnector` class.

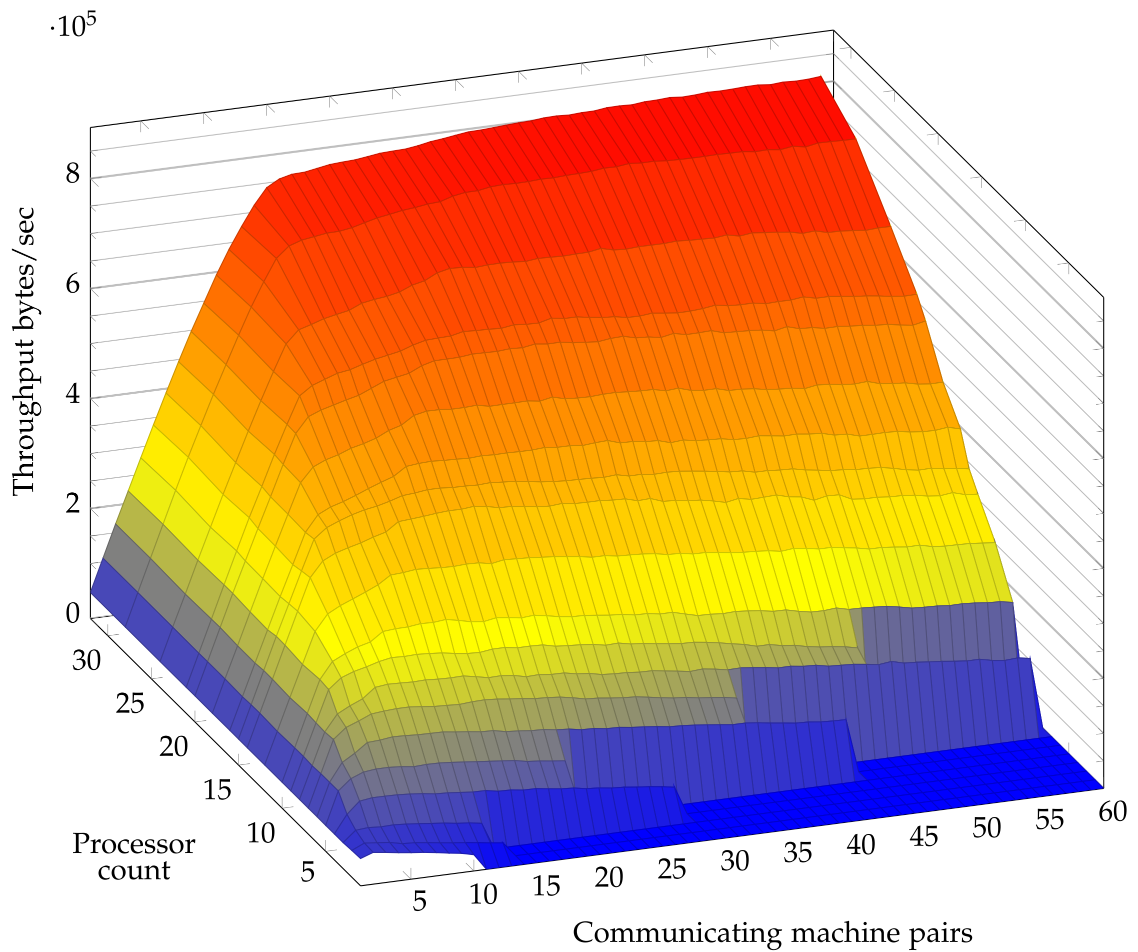


Figure 5.29: *A surface plot of all combinations of machine pair count and processor count for the `SpeedTest` application on the Blueshell target. A 500-byte message size and the sequential processor allocator were used.*

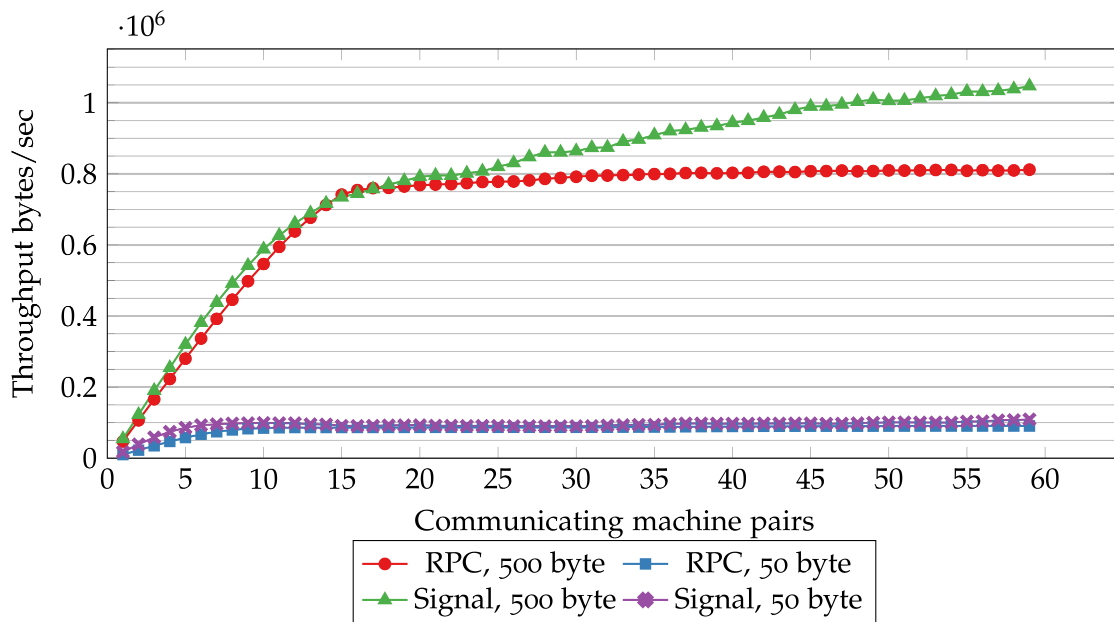


Figure 5.30: *A comparison of the **RemoteProcedureCall** (RPC) based SpeedTest implementation and the **Signal** based implementation for both 50 and 500-byte message sizes.*

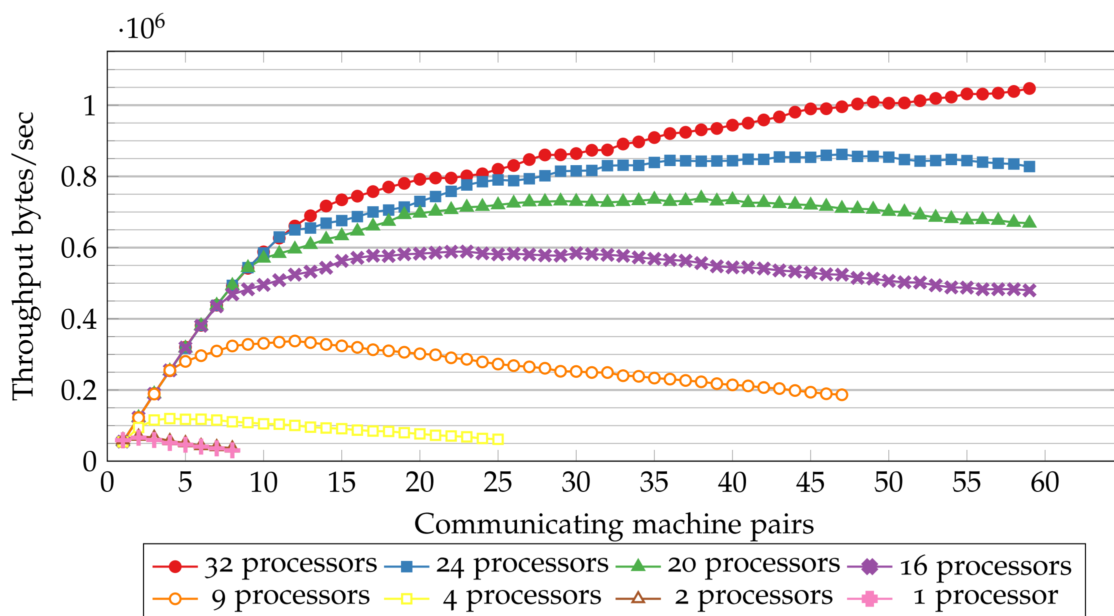


Figure 5.31: *The communications throughput of the **Signal**-based SpeedTest implementation across various sizes of network.*

Figure 5.30 provides a comparison of the **RemoteProcedureCall** based SpeedTest application, and a reformulation using **Signal** channels, which are unidirectional, destructive, and single-buffered (see section3.2.5.3). This version ('SpeedTestHP'[18]) has a true mutual dependency between its client and server machines. Each server has a reference to its particular client and responds only to that client on receipt of a new query. As this application uses destructive communications the Machine Java framework does not need to perform any handshaking between machines to send data.

It can be seen in figure 5.30 that when using large messages the overall throughput is greater for the **Signal**-based implementation at high machine counts. A more complete picture of **Signal** performance is provided in figure 5.31 where it can be seen that the variation between high machine-count throughput is more pronounced than the RPC version.

5.4.3.1 Summary

In this section the communications overheads of a Machine Java application were investigated with a variety of factors found to affect overall messaging performance:

- The target architecture (somewhat obviously) impacts communications performance as this defines the processor and network types, and the operating frequencies that all components execute at.
- The number of processors in the platform.
- The number of machines attempting to communicate.
- The exchanged message size.
- The specific allocation of machines to processors.
- The channel protocol used to communicate.

It can be seen that for the sizes of network considered in this section message throughput is limited by available processing capacity. There were not enough processors even in the largest instances to observe apparent limitations due to underlying network capacity. Overall, the implementation of Machine Java presented in this thesis

[18]The client and server machine implementations are provided in appendix C.2.

has substantial messaging overheads which in turn limits applications to low messaging throughput. Future implementations would benefit greatly from compiler-assisted or hardware-accelerated serialisation; messaging performance could be dramatically improved if a processor did not have to actively serialise and deserialise all Java objects.

As neither the application model nor Machine Java introduce any non-application dependencies between processors, it can be expected that applications should scale gracefully onto considerably larger platforms; whole-application throughput can be expected to increase with additional processors until the communications substrate itself becomes the limiting factor. The results presented in this chapter do not invalidate this expectation but also do not *prove* it; the linear scaling region where adding machines resulted in a proportionate increase in total throughput is reassuring but not a guarantee of the performance on much larger platforms. On platforms where the network capacity is the limiting factor the communications patterns of applications and the internal messaging used by the non-destructive channels seem likely to introduce emergent behaviour. The SpeedTest microbenchmark itself will also eventually encounter scaling issues even on indefinitely large platforms due to the tiny but finite overheads of communicating with the central `SpeedTest` machine which aggregates results.

5.4.4 Computation Performance

Evaluating the computational overheads and performance is less involved than for communications overheads as computation within a machine does not interact with any other machines. The computational throughput of an individual machine on an unloaded platform will simply reflect the capabilities of the processor and of the Chi (see section 4.7) compilation strategy. For embarrassingly parallel workloads (those in which the subdivisions of a task can be worked on independently) Machine Java can be expected to fully exploit all of the processing capacity available in a platform with no processor-count dependent overheads; if the processor count is multiplied by five, then five times the computational throughput can be expected. This section aims to verify this hypothesis using the DistributedMD5 application discussed in section 5.4.2.4. This microbenchmark represents an application where the only communications are used to report progress and are not essential to the workload. Although simple, this workload is representative of the *proof-of-work* algorithms used for spam-email prevention [58] and cryptocurrency 'mining' [140, 25].

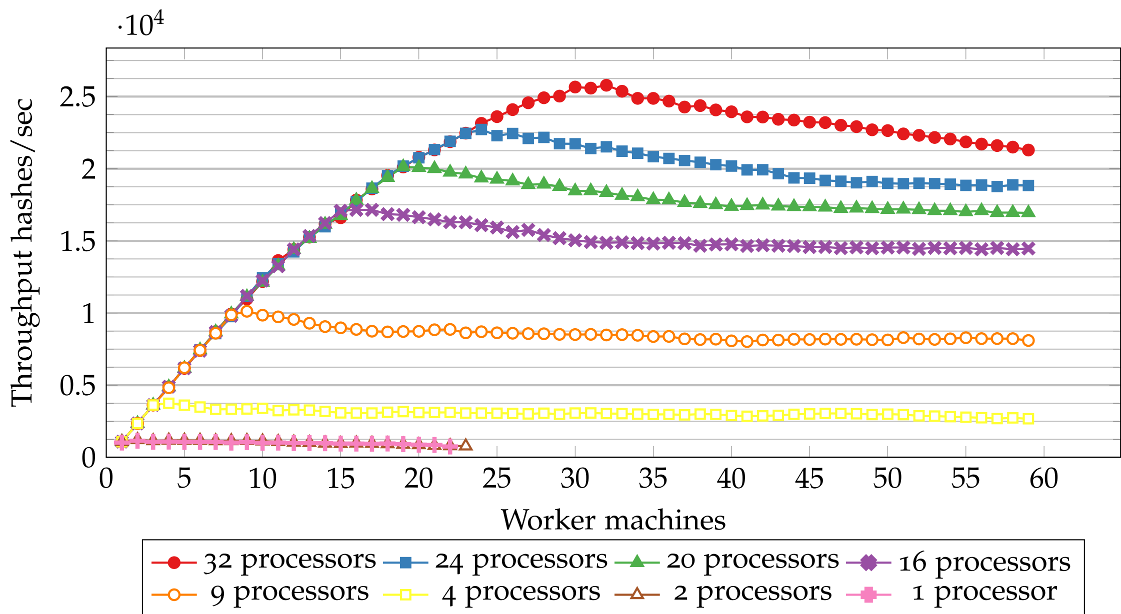


Figure 5.32: *The overall computational throughput of the DistributedMD5 application on a Blueshell target with various processor counts. Performance is measured by the number of cryptographic hashes computed per second across all machines.*

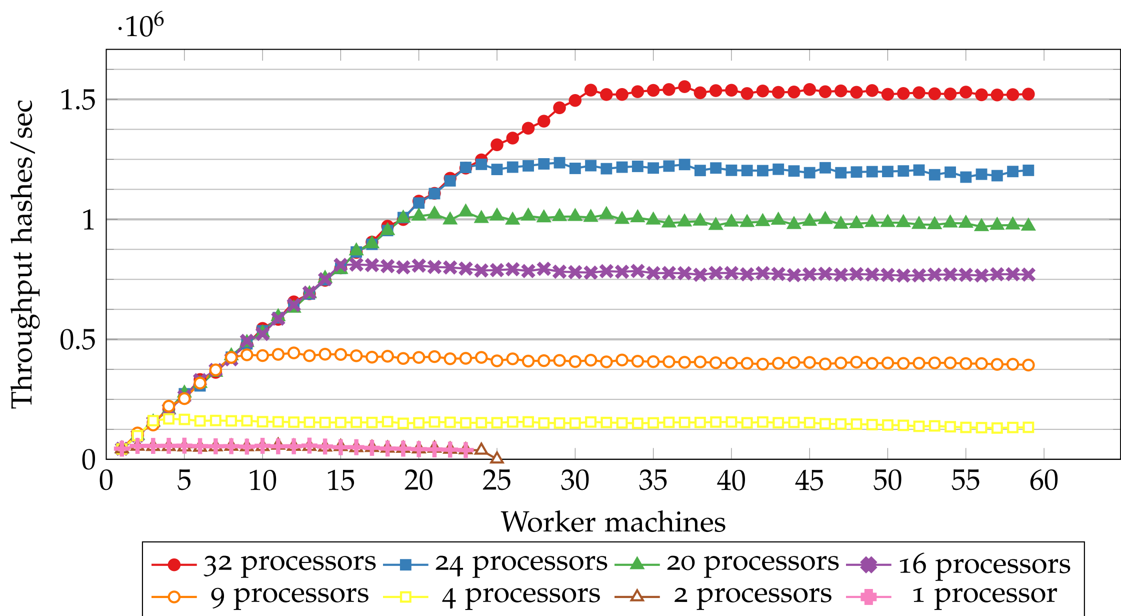


Figure 5.33: *The overall computational throughput of the DistributedMD5 application on an OS target with various processor counts. See figure 5.32 for comparison with the Blueshell platform.*

Figure 5.32 presents the throughput of the DistributedMD5 application on Blueshell networks with varying processor counts. For contrast the execution of the application for the OS target is also provided in figure 5.33. The application throughput is measured as the total number of cryptographic hashes computed by all machines per second. A number of observations can be made from figures 5.32 and 5.33:

- On the OS target the total application throughput is approximately proportional to the number of utilised processors, whereas on the Blueshell target performance appears to improve less than linearly above approximately 20 machines. The diminishing returns from adding more machines on Blueshell, even when there are unused processors indicates some interference between the active machines. As the application is known (by construction) to avoid communication this is a strong indication that competition for code memory bandwidth must be beginning to impact performance. This is because code memory bandwidth is the only shared resource used by all active processors.

- When the number of machines is less than the number of processors in the platform, the plot is unaffected by the processor count.

- On both targets the application's throughput reaches a maximum when the machine count is equal to the platform's processor count.

- After the machine count has exceeded the processor count there is a significant observable decline in throughput on the Blueshell target, and especially for the 32-processor platform. This is an indication that non-application code (which does not contribute to measured performance) required to multiplex machines on a processor is a considerable expense on Blueshell. However, on the OS target there is a negligible decline in performance after machine count exceeds processor count. In combination this is more evidence that limited instruction cache on the Blueshell's MicroBlaze processors is impacting performance. The OS target processors can easily contain the entire application in the instruction cache, but the MicroBlaze processors only have 8KiB of instruction cache each. The `MD5Worker` machine and MD5 computation library require a total of 8,112 bytes of code memory. Even with a perfect mapping from code memory into processor caches[19], the `MD5Worker` can-

[19]This is highly unlikely as the MicroBlaze tile's cache is direct-mapped.

not quite fit into the instruction cache. When there are more machines than processors there must be some processors with multiple machines and this implies some effort in the Machine Java runtime to decide how to multiplex the processor's time. This will necessarily require some code memory which must be fetched from the read-only shared memory assuming the processor's instruction cache has been filled by the `MD5Worker`'s computational kernel.

Competition for limited code memory bandwidth not only explains the diminishing returns above 20 processors on Blueshell, but it also explains why the high processor-count platforms appear to suffer more after the machine count exceeds processor count: there are more processors competing for the same memory bandwidth.

- This experiment (in addition to the SpeedTest experiments) demonstrates that the artificial memory capacity constraints when compiling for the OS target are a faithful representation of the true memory constraints on the Blueshell target. On one and two processor platforms the application fails due to memory exhaustion (`OutOfMemoryErrors`) at similar machine counts. The exact timing of the failure is not identical but it is sufficient to determine the basic plausibility of an application in a resource constrained context even if real hardware were unavailable.
- It can be seen that one and two processor platforms perform the same. This is also shown in the SpeedTest experiments and is a result of the `ProcessorManager`'s implementation. `ProcessorManager` only choses to allocate new machines onto the current processor if it has no alternative, so on a two processor platform only the start machine will exist on the origin processor and all subsequent machines allocated by the start machine will be requested from the second processor.

These observations lead to two further questions:

1. As code memory bandwidth contention is strongly implicated for the failure to achieve linear performance scaling on the Blueshell target, would larger processor caches or a smaller computational kernel enable linear performance growth to be realised?
2. The experiments indicate that there are some hidden but non-trivial computational overheads due to the framework. What is the extent of these overheads?

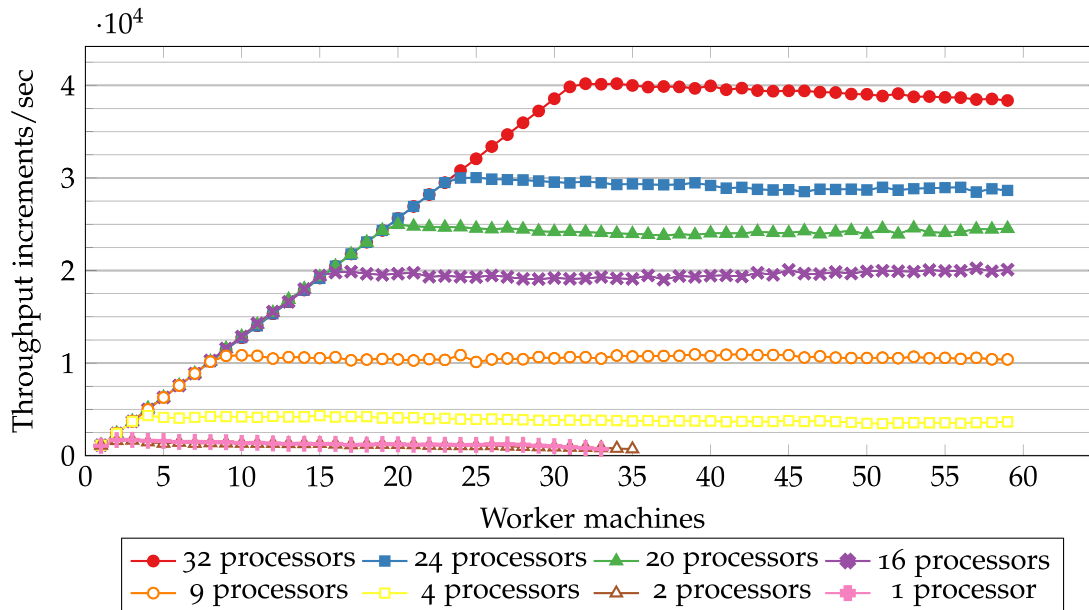


Figure 5.34: *A comparison of the throughput for a minimal computational kernel on Blueshell. The kernel is so small that effects due to instruction memory contention can be discounted. Contrast with figure 5.32.*

For the purposes of this thesis, the MicroBlaze instruction caches are fixed at their maximum size. There are not enough block RAMs available in the host FPGA to provide larger instruction caches. This is not a substantial problem as a smaller computational kernel can be used instead. The specific performance numbers from a different kernel are incomparable to the numbers in figure 5.32, but the *shape* of the plots acquired will indicate if a reduced kernel size has eliminated inter-processor interference. Figure 5.34 provides the results from a modification of DistributedMD5 with a vastly reduced kernel size. In this application each worker machine only increments a long integer before it uses a `Yield` to allow another event handlers a chance to execute. This modification of the application only requires 1,288 bytes of code memory[20] for its worker machine, and substantially less than this is actually used to perform its 'work'. It can be seen in figure 5.34 that reducing the computational kernel to much less than the instruction cache size has enabled linear computation scaling on the Blueshell platform. Some performance degradation after the machine count exceeds the processor count is still evident but it is now more similar to the gentle decline observed on the OS target. It can also be observed that the reduced dynamic memory requirements of the smaller kernel allows

[20]DistributedMD5 requires 8,112 bytes for `MD5Worker` and the MD5 class combined.

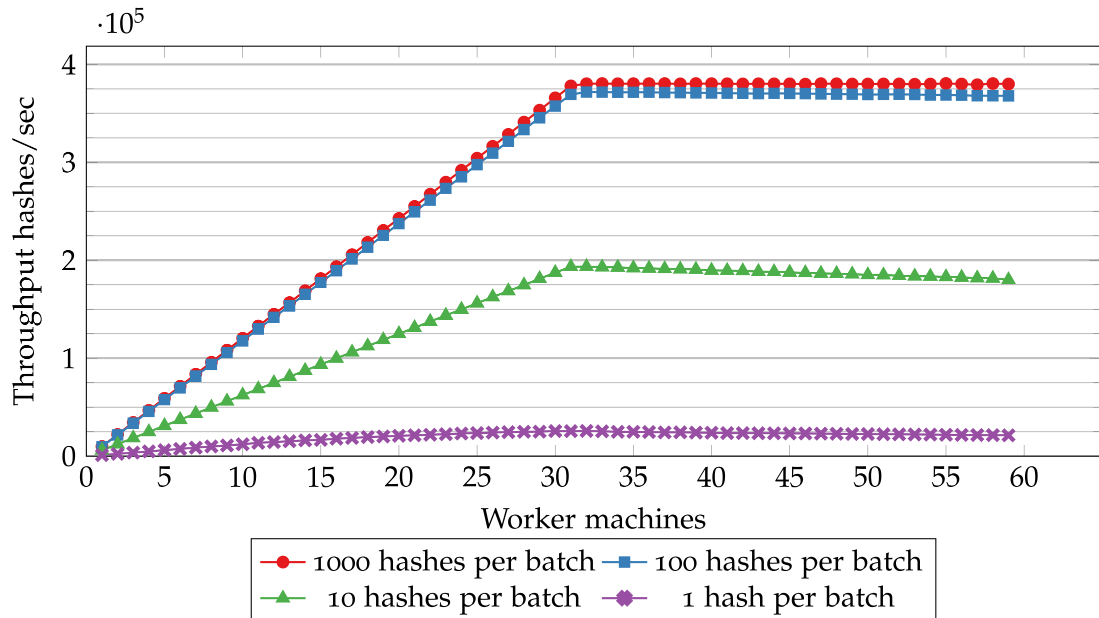


Figure 5.35: *The impact of batching computation on application throughput on a 32-processor Blueshell platform. The batch size is the number of hashes computed before the event handler uses a* `Yield` *(see section 3.2.3.2) to return control to the Machine Java framework.*

for more machine instances on small platforms before the memory is exhausted.

Question 2 concerns the framework overheads rather than multiprocessor scaling, but this is largely hidden in the previous experiments. The presence of some overhead due to the framework can be inferred from the decline in performance after machines become overmapped but it is certainly not trivial to calculate what this overhead might be.

As the Network-Chi implementation of Machine Java is non-preemptive at the processor level there can be confidence that if application code is executing, not only is no other application code also executing, but also no framework code is active. This allows framework overhead to be measured by varying the amount of work performed before control is released by application code. In figure 5.35 the DistributedMD5 application is executed with a number of different 'batch-sizes'. In this version a `MD5Worker` computes a batch of hashes before yielding. It can be seen from the plots that the previous experiments were actually *wildly* inefficient. Performing ten computations in a batch provided an approximately eight-fold improvement in application throughput, while moving to 100 or 1,000 computations per batch provides an approximately fifteen-fold improvement in throughput. It can be seen that there is a negligible improvement in batches of 1,000 compared to 100 for this application.

An attempt can now be made to quantify the Machine Java runtime overheads: Given the observed diminishing returns above batch sizes of 100, it can be assumed for the sake of simplification that the framework overhead is negligible at a batch-size of 1,000. This demonstrates that a 32-processor Blueshell can perform approximately 380,000 hashes per second. As the non-batched application only performs approximately 26,000 hashes per second there is the equivalent of 354,000 hashes of computation wasted: 93% inefficiency. This can alternatively be viewed as a cost of approximately 14 hash-equivalents of work per yield to the framework.

The minimised kernel considered in figure 5.32 is at least two orders of magnitude smaller than the MD5 kernel so could be expected to require batch sizes of at least 10,000 or likely 100,000 to achieve high efficiencies.

5.4.4.1 Summary of Computation Performance

In this section the computation overheads of a Machine Java application were investigated. The primary findings were:

- Machine Java appears to enable the scalable exploitation of communication-centric hardware architectures. This was demonstrated by the results where computational throughput was linearly related to the number of processors in use.
- Competition for instruction memory bandwidth was the primary constraint on scalability in the experiments.
- The framework overhead was found to be approximately equivalent to the work required to compute 14 MD5 digests of a 64-byte buffer per event handled, although it is difficult from these results to separate the effects of inadequate instruction caching from the work actually required by the Machine Java runtime. On Blueshell 14 MD5 digests is approximately 1.2ms. This overhead includes the time taken by Machine Java to decide which event handler to execute next and the time taken to perform background housekeeping tasks.

The experiments provide more evidence that shared memory can easily introduce performance bottlenecks, even in situations where the memory is read-only. A number possibilities for mitigating the impact of limited instruction memory bandwidth were identified:

- Reduce the size of the computational kernel so that it is more likely to fit into processor-local instruction cache.
- Batch computations together into a single event handler. This has the effect of reducing the time required by Machine Java to determine the next activity, which in turn reduces the amount of different code executed so increasing the effectiveness of a limited instruction cache.
- Increase the size of processor-local instruction caches or use processor-local memory for intensively used computational kernels. This point could be considered vacuous as the platform is almost certainly not under an application designer's control.

5.5 Summary

In this chapter the Machine Abstract Architecture was evaluated for its effectiveness via the Machine Java reference implementation. Overall the feasibility of programming resource constrained MPNoCs using a machine-oriented programming model was validated: Machine Java is capable of expressing complex and dynamic applications which are compact enough to execute even on substantially limited processors. While the implementation was demonstrated to be sufficiently compact, it was also shown that Machine Java applications were portable across a variety of platforms: simulated mesh networks in a JVM and on Linux, and true mesh networks on an FPGA-realised MPNoC. No modifications to application code were required to accommodate changes in network size or underlying instruction set architectures.

The communications throughput achievable by a Machine Java application was shown to be affected by multiple factors both from the application and also the platform. The number of processors and the number of machines have an obvious and substantial impact on performance, but the impact on the framework's allocation of machines to processors has a far more subtle effect. Overall communications performance was shown to be quite limited and most significantly constrained by the processing effort required to send and receive messages.

Computational throughput of Machine Java applications was shown to scale well with respect to the number of processors available in the platform. On the hardware evaluation platform limited instruction memory bandwidth was shown to significantly limit the scalability of applications where the computational kernel was too large to fit into processor-local instruction caches. Machine Java framework overheads were shown to be substantial but also avoidable: simple work-batching greatly improved execution efficiency.

Java's runtime safety features were shown to have a profound impact on static memory consumption which could be avoided by disabling these features when confidence in an application's correctness has been achieved. After all minimisation techniques are applied the single largest contributor to an application's static memory consumption is the string literals embedded in the code. This represents a significant opportunity for further reductions in application size in future.

Programming with Machine Java is shown to avoid some of the conventional pitfalls of concurrent programming: data races and deadlocks are far less problematic as

data cannot be shared. However, machine-oriented programming is not a magic-bullet and concurrency-related application design errors are still very possible. Application livelock is shown to be possible when an application makes intuitive but invalid assumptions about intermachine message arrival ordering.

Finally, the ability construct applications with simple static structure but complex self-elaborated runtime structure was discussed. This pattern provides the advantages of reduced static memory consumption and reduced apparent application complexity, but increases difficulty of extracting runtime structure from the application definition.

Chapter 6

Conclusion

Contents

5.1 Overview . . . 224
5.2 Hardware Evaluation Platform . . . 226
5.2.1 The Blueshell NoC Generator . . . 226
5.2.2 Evaluation Platform Specification . . . 230
5.3 Programming with Machine Java . . . 232
5.3.1 Defining, Extracting and Fixing Application Structure . . . 233
5.3.2 Application Platform Independence . . . 246
5.3.3 Complex and Dynamic Application Structures . . . 249
5.3.4 Concurrency Without Sharing . . . 254
5.3.5 Fault Tolerance in Machine Java . . . 262
5.4 Overheads and Scaling . . . 265
5.4.1 Static Memory Consumption . . . 265
5.4.2 Experimental Methodology . . . 276
5.4.3 Communications Overheads . . . 284
5.4.4 Computation Performance . . . 298
5.5 Summary . . . 306

6.1 Revisiting the Hypothesis

In this thesis *machine-oriented* programming was presented as an approach for programming highly multiprocessor but also constrained MPNoC architectures. This work is motivated by the observation that such architectures will be challenging or impossible to program with conventional thread-parallel models.

As discussed in section 1.5.1, the work in thesis is intended to test the following hypothesis:

Resource constrained multiprocessor networks on chip with non-coherent caches can be programmed effectively with a general purpose programming language through the application of an actor-oriented model of concurrency.

Several success criteria can be extracted from the hypothesis and considered against the work individually:

resource constrained MPNoCs The application model described in section 3.2 enables communication-centric software to be independent of an implementation platform's topology. Not only does this mean that applications can be supported on MPNoC architectures, but the application does not even need to be aware of the hardware's design pattern. The assertion that the model enables application-platform independence was considered in section 5.3.2, where it was found that applications can be *mostly* platform independent but non-functional goals are difficult to achieve in the absence of any platform details.

The resource constraints of the target domain was primarily addressed by the implementation methodology described in section 4.7. This compilation approach was found (section 5.4.1) to effectively minimise the post-compilation requirements of Machine Java (chapter 4) application code. Execution of the complex Machine Java reference implementation was demonstrated (section 3) on processors with only 48KiB of memory available for the Java heap.

effective programming The 'efficacy' of the programming model is judged to be a success based on the following decomposition:

- Viability: The executable Machine Java programming framework validated the feasibility of the machine-oriented application model.
- Supports software engineering: The use of the Java programming language enables existing software engineering practices and ecosystem to be applied to machine-oriented programming.
- Provides some benefit: Multiple benefits of varying importance have been identified:
  - The ability to program architectures in the target domain with Java (chapter 4)
  - Extended analysis of program structure (section 5.3.1)

  - Exploitation of the benefits of a programming model with actors (sections 5.3.4 and 5.3.5).
  - Results gathered indicate that the programming model is likely to scale to large target platforms; no programming model related overheads that are a function of the platform's processor count were identified (section 5.4).
- Tolerable costs: The costs of implementation were found to be high in some situations (section 5.4) but not so high that the approach can be considered generally impossible. However, the overheads that can be tolerated depend on on the ultimate purpose of a system rather than the programming model.

general purpose Neither the model nor the demonstrated realisation of Machine Java limit the scope to a specific application domain.

actor modelled concurrency The machine-oriented application model described in section 3.2 is certainly an actor-oriented model of concurrency: Machines *are* actors.

Overall, the thesis hypothesis is considered to be substantially validated: Resource constrained MPNoCs *can* be exploited with general purpose programming languages.

The remainder of this chapter contains a summary of the contributions of this thesis in section 6.2. A selection of open questions raised by this research are provided in section 6.3 and finally the thesis is concluded in section 6.4.

6.2 Thesis Contributions

The significant contributions of this thesis are as follows:

6.2.1 Machine-Oriented Programming Model

The programming model described in section 3.2 provides a rich *machine-oriented* approach to design of concurrent applications. Actor-oriented models of concurrency promise a degree of simplicity and fault tolerance that are retained in this thesis' machine-oriented application model. The application model presented explicitly avoids the introduction of centralised coordination that would inhibit application scalability, and improves application expressiveness by permitting (and defining the consequences of) multiple channels per machine with potentially diverse communications protocols.

This model is language agnostic and can be applied to a wide variety of practical programming languages. The restrictions imposed by the model enhance the platform independence of applications and enable application structures and properties to be extracted regardless of the implementation language.

6.2.2 Machine Java Framework

An early version of this framework was published previously [162].

A reference implementation of the application model and the other elements of the Machine Abstract Architecture (see chapter 3) was presented. *Machine Java* enables machine-oriented applications to be designed in the very widely used Java programming language. The use of Java validates the applicability of the model to a real and practical programming language while also enabling the re-use of a large ecosystem of existing methodologies and tools for software development.

Machine Java demonstrates compile-time enforced type-safety across an entire distributed application, and Java's type system is further used to statically enforce the consistency between machines of communications protocols. Applications and Platforms are both defined exclusively by their Java code, reducing the possibility of errors due to inconsistent specifications.

The communications protocols described in the application model (section 3.2.5) are provided in Machine Java enabling applications to select channel characteristics that are appropriate for their specific use-case. Machine Java does not attempt to hide the distributed nature of an application nor attempt to corral all intermachine communications into a single category.

A strategy for the execution of Machine Java applications on a standard Java Runtime Environment was presented enabling rapid behavioural prototyping of application software.

XYNetworkPlatform

A basic Machine Java platform was presented to enable execution on homogeneous meshes of processors addressed by their X and Y coordinates. The *XYNetworkPlatform* demonstrated the principle of platform-directed machine allocation via processor iteration, and some of the impact this can have on runtime performance.

The XYNetworkPlatform provided a bridge between the Machine Java specification

and a runtime that uses the mesh-networking abstractions provided by Network-Chi to implement its communications. This platform example demonstrates how applications are isolated from the topology of the runtime environment, and how a Machine Java platform model can actually be used to target many distinct implementation (hardware) architectures.

6.2.3 Chi Optimising Java Compiler

Published previously [163, 164, 165].

An effective strategy for the compilation of Java applications into minimal binaries was presented (see section 4.7). The *Chi* compilation approach minimises code requirements through extensive pruning of used Java code and ahead-of-time compilation of the application against the JVM's behaviour. In many cases runtime behaviour can be determined at compile time (such as which method body will be invoked due to a virtual invocation) so substantial savings are achieved.

A minimal in-memory representation of Java objects enables processors with very limited RAM to be targeted, and a garbage collection profile was determined that minimises the runtime memory wastage while maintaining acceptable application throughput.

A compiler assisted low-level mesh networking abstraction was introduced to Java, enabling identical Java code to execute in a JVM, on POSIX operating systems without Java, or on the Blueshell evaluation hardware without any host operating system.

6.2.4 An Empirical Evaluation of Machine-Oriented Programming

An empirical evaluation of Machine Java, the compilation strategy and designing applications according to a machine-oriented model was provided in chapter 5. This evaluation demonstrated the viability of Machine Java applications even in the context of highly constrained processors (section 5.4).

This validates the programming of communication-centric MPNoC architectures in Java using a machine-based model of concurrency, which is an important result as these architectures are not programmable using standard thread-based models. The evaluation of overheads and scaling in section 5.4 provides guidance for design decisions when constructing a machine-oriented application.

6.3 Future Work

The work presented in this thesis creates the opportunity for many different avenues of future research:

6.3.1 Exploitation of Extracted Application and Platform Information

Although the work in chapter 5 considered the extraction of graphs from a Machine Java application these graphs were not put to any use by the presented framework or compiler. More sophisticated implementations could consume compiler-generated graphs, or even graphs extracted from a system at runtime (such as figures 5.26 and 5.27 on pages 292-293) to enable any of the following:

better machine placement such as allocation of machines into physically local areas if it can be determined that they will communicate frequently. Runtime profiling of applications would be particularly effective for this task.

proactive load balancing by **ProcessorManager** machines would enable allocation of dynamic machine workloads to a subset of processors without relying resource exhaustion of remote processors to drive the allocation process.

non-uniform architectures such as those with heterogeneous processor architectures or irregular topologies are difficult to exploit without information about the application's machine resource requirements. **ProcessorManager** machines and the platform model could use application resource-requirement information to make more intelligent allocation requests.

6.3.2 Garbage Collecting Machines

It was identified in chapters 3 and 4 that machine end-of-life is particularly problematic: Machines cannot be safely destroyed on request as there is no way to guarantee that the machine is no longer in use by some other machine.

Machine Java will destroy and reclaim resources used by machines in a limited selection of circumstances (see section 4.3.5), but for ordinary machine definitions this will only happen after the application code has explicitly *closed* the machine's inbound channels, and only destructive channels even have the opportunity to be closed.

Further exploration of application design using the concept of simple ephemeral machines (section 4.3.5.1) may be able to identify techniques by which familiar programming patterns can be expressed using SEMs, but the problem of ordinary machine end-of-life remains.

A mechanism that could identify machines, even those with non-destructive channels, that are universally unused would be of great use to future implementations. An attractive starting point would be to consider the substantial literature that already exists in the field of distributed garbage collection [161] and in particular the strategies in use by other distributed actor-oriented frameworks.

6.3.3 Relocating Active Machines

Machine Java, as described in this thesis, does not support the migration of machines from one processor to another. This activity is compatible with the application model as machine instances are unaware of the physical placement of their own and other machines. If a framework supported the relocation or migration of live machines this would enable applications to respond (possibly transparently) to dynamic changes in the platform's structure. Such changes might be the result of hardware failures, the 'hot' addition or removal of processors. Even on a single chip it may be possible for processing units to come and go over time if the platform is implemented on a *partial dynamic reconfiguration* (PDR) capable FPGA.

The ability to migrate live machines would also enable dynamic adjustment to prevailing environmental conditions. If a system has run critically low on battery, it might be desirable to migrate some machines to low power processors rather than halt the system altogether.

However, relocating a live machine represents a variety of complex technical challenges:

- Machines reference one another via ordinary Java object references, and Machine Java internally uses identifiers for machines that contain the identity of the host processor. If a machine were migrated to a new processor all of the machine references would no longer be valid. Some mechanism must be identified to dynamically update machine references to the new location of a migrated machine.

- Moving a machine that is active it challenging itself. The execution must be

stopped, state moved and execution resumed elsewhere:

  – Halting a tasks execution and saving the *context* for later is a common activity for an operating system, but on processors without an operating system this may be more challenging.
  – A machine's internal state may not have the same representation (byte ordering and floating point format, for example) on the source and destination machine, necessitating the ability to translate the state. Furthermore, Java object references will need to be adjusted if the in-memory representation is a pointer, as it is when compiled with Chi.
  – Resources in use by a machine may have internal state that is left inconsistent following the machine migration. This may also represent a security and privacy concern for a system.

- Automatic migration not triggered by hardware failure could degrade overall performance rather than improve it.

6.3.4 Communications Performance Improvements

The primary cause for concern highlighted in the evaluation is the poor communications throughput achieved (see section 5.4.3). Applications that are dependent on exchanging non-trivial quantities of data would benefit substantially from higher performance communications. The issue is the poor efficiency of the object serialisation mechanism (described in section 4.7.3.3). Identifying more efficient, and particularly hardware assisted Java serialisation would enable the processor to continue with useful work providing benefits for computational as well as communications throughput.

Some platforms will have shared memory available, and this might enable copy-free communications but the current Machine Java runtime is not able to take advantage of this as objects are allocated into processor local memory which may not addressed by remote processors. The object would have to be allocated into shared memory initially or somehow be migrated in memory.

Where shared memory is available, it may be possible to enable the communication of mutable objects without violating intermachine isolation by controlling the *aliasing* of the objects to be communicated. Work has been done on the representation of aliasing rules within a type systems [57], and in particular so-called *isolation* types have been

demonstrated [194] which can ensure the single-ownership of messages in an actor-oriented context. The application of these techniques would enable the performance penalties associated with repeated construction of immutable Java objects to be avoided, but they may not be appropriate in MPNoCs without shared memory.

6.3.5 Multiple Applications

Allowing multiple applications to communicate remains a challenge. The application model as defined in section 3.2 allows for multiple applications to co-exist on a platform but there is not any mechanism for them to communicate as no machine can have a reference to the machines in the other application. There are two promising approaches:

1. Define some mechanism to allow inter-application communication of Machine references within the machine model. This could plausibly be supported by enabling the `ProcessorManager` machines (of which all applications always have references to, see section 3.4.1) to act as gateways between the applications. An application could lodge a reference to some 'application host' machine with a selected `ProcessorManager`. This may then be looked up by other applications at runtime. There are two key difficulties with this approach:

   (a) How could an application know which `ProcessorManager` to query for a reference to another application at runtime? Applications do not have any concept of the platform topology so cannot in general know what `Processor Managers` are available, much less chose which one to query. Even if a strategy for locating `ProcessorManagers` is defined in a platform independent way, scaling this lookup procedure to networks of thousands or millions of processors is likely to be challenging.

   (b) The ability to obtain a reference to a separate application at runtime could undermine the static determination of the possible communications paths between machines, which is a key advantage of the machine model. Frameworks (such as Machine Java) might chose to facilitate this by requiring that the source for the referenced application, or the source of a contract for the application (such as abstract machine class definitions) are available at compile time.

2. Some mechanism for address-based communications could be provided. In this

scheme it would be possible to 'lookup' another application via an address which has a lifetime that exists beyond the lifetime of the other application. Such an address could be IP-style [168] or a URL [27]. Schemes with long lived addresses have all of the complications listed above, and many more of their own including the possibilities of invalid addresses, valid addresses with missing destinations and resolution of addresses to references.

Another approach is to accept that separate applications must resort to use of traditional (or platform defined) interprocess communication, such as sockets, signals or named pipes. This solution is as practical as it is unattractive. It could clearly be facilitated on any platform without the need to define a new application model, but none of the safety or analysis advantages of the existing machine model would be gained across applications.

6.4 Epilogue

The era of *difficult* silicon is coming: Chips with ever more processing units, with communication-centric architectures, with heterogeneity which cannot be ignored in software, and most critically with asymmetric memory. Appropriate programming models will become more important than ever if such architectures are to be efficiently harnessed. Actor-oriented models of concurrency appear to provide such an excellent match for communication-centric hardware such as MPNoCs, that the challenge becomes establishing what is required for these models to achieve acceptance. The work presented in this thesis is one data-point on the long path towards an acceptable programming model for these difficult future architectures.

Appendix A

Machine Java API Class Examples

Contents

6.1 Revisiting the Hypothesis . . . 309
6.2 Thesis Contributions . . . 311
6.2.1 Machine-Oriented Programming Model . . . 311
6.2.2 Machine Java Framework . . . 312
6.2.3 Chi Optimising Java Compiler . . . 313
6.2.4 An Empirical Evaluation of Machine-Oriented Programming . . . 313
6.3 Future Work . . . 314
6.3.1 Exploitation of Extracted Application and Platform Information . . . 314
6.3.2 Garbage Collecting Machines . . . 314
6.3.3 Relocating Active Machines . . . 315
6.3.4 Communications Performance Improvements . . . 316
6.3.5 Multiple Applications . . . 317
6.4 Epilogue . . . 318

Representative examples of Machine Java classes are reproduced here for reference including the most complex of the channels (`RemoteProcedureCall`) and time event sources (`Period`).

A.1 RemoteProcedureCall.java

```
1  package mjava.core.tpif;
2
3  import java.io.IOException;
4
5  import mjava.core.Handler;
6  import mjava.core.Machine;
7  import mjava.core.runtime.drivers.PreviousDatumNotAcceptedException;
8  import mjava.core.runtime.drivers.TPIFDriverRx;
9  import mjava.core.runtime.drivers.TPIFDriverTx;
10
11 /**
12  * Root of all bidirectional protocols.
13  *
14  * Event handlers are provided with a bidirectional envelope
15  * which can be used to reply to the querying machine without knowing
16  * what their type is.
17  *
18  * @author gary
19  *
20  * @param <Q> Query type
21  * @param <R> Response Type
22  */
23 public class RemoteProcedureCall<Q, R> extends TPIFBidirectionalProtocol
       <Q, R> {
24
25   protected final int bufferLength;
26
27   /**
28    * A connector for querying remote procedure call protocols. This,
          like other connectors should be shutdown after its last use.
29    *
30    * @author gary
31    *
32    * @param <Q> Query type
33    * @param <R> Response Type
34    */
35   public static class RemoteProcedureCallConnector<Q, R> extends
         TPIFBidirectionalConnector<Q, R> implements Runnable {
36
37     protected final RemoteProcedureCall<Q, R> host;
38     protected final TPIFDriverTx<BidirectionalQuery<Q, R>> txDrv;
39     protected final TPIFDriverRx<R> rxDrv;
40     private boolean ephemeral;
41
42     /**
43      * Constructs a new RPC connector. The reply handler must not be
            null!
```

```
44       * @param host The host RPC protocol.
45       * @param replyHandler The handler to deal with replies.
46       */
47      public RemoteProcedureCallConnector(RemoteProcedureCall<Q, R> host,
            Handler<Envelope<R>> replyHandler) {
48        super(replyHandler);
49        this.host = host;
50        //No unblocked handler is set as we don't actually care when this
             is unblocked.
51        // sending will be forbidden until a response is received from the
              server.
52        // (and this can only have happened after an unblocked would have
             been received.)
53        txDrv = Machine.getThisMachine().getMachineDriver().
             getTPIFDriverTx(host.getExternalID(), false, true, null);
54        //replies come to this driver: (keep exactly one slot for replies,
              destructive protocol: no handshakes!)
55        rxDrv = Machine.getThisMachine().getMachineDriver().
             getTPIFDriverRx(this.getExternalID(), this, true, 1);
56        rxDrv.accept(); //be ready for the first reply.
57      }
58
59      /**
60       * Sends a query to the RPC.
61       * Blocks until there is data available and returns it.
62       * Only use for non-event based, one-shot queries.
63       *
64       * @param query The query to send.
65       * @throws PreviousDatumNotAcceptedException If an attempt is made
            to send another query before a reply has been received for an
            existing in-flight query.
66       */
67      private R sendAndWait(Q query) throws
            PreviousDatumNotAcceptedException {
68        //Wrap the query with this connector's external address for
             replies.
69        send(query);
70        return rxDrv.getBlocking();
71      }
72
73      /**
74       * Sends a query to the RPC.
75       * @param query The query to send.
76       * @throws PreviousDatumNotAcceptedException If an attempt is made
            to send another query before a reply has been received for an
            existing in-flight query.
77       */
78      public void send(Q query) throws PreviousDatumNotAcceptedException {
79        //Wrap the query with this connector's external address for
```

```
        replies.
80      //if (Network.amOrigin()) System.out.println("RPC: sending query")
            ;
81      txDrv.send(new BidirectionalQuery<Q, R>(getExternalID(), query));
82    }
83
84    /**
85     * A synonym for send.
86     * @param query
87     * @throws PreviousDatumNotAcceptedException
88     */
89    public void query(Q query) throws PreviousDatumNotAcceptedException
         {
90      send(query);
91    }
92
93    /**
94     * Executed by rxDrv to indicate a new datum has arrived.
95     */
96    @Override
97    public void run() {
98      registerEvent(new TPIFEnvelope<R>(rxDrv, this, ephemeral));
99    }
100
101   public void shutdown() throws IOException {
102     rxDrv.shutdown();
103     txDrv.shutdown();
104   }
105
106   public RemoteProcedureCallConnector<Q, R> setEphemeral(boolean
         ephemeral) {
107     this.ephemeral = ephemeral;
108     return this;
109   }
110
111 }
112
113 /** the driver that enables this RPC to work. This driver receives
       queries. */
114 protected TPIFDriverRx<BidirectionalQuery<Q,R>> _drv;
115
116
117
118 public RemoteProcedureCall(Handler<? super ReturnableEnvelope<Q, R>>
       queryHandler) {
119   this(BoundedBuffer.DEFAULT_BUFFER_LENGTH, queryHandler);
120 }
121
122 public RemoteProcedureCall(int bufferLength, Handler<? super
```

```
      ReturnableEnvelope<Q, R>> queryHandler) {
123     super(queryHandler);
124     if (queryHandler==null) throw new NullPointerException("queryHandler
           may not be null.");
125     this.bufferLength = bufferLength;
126     setupDriver(bufferLength);
127   }
128
129   /**
130    * Sets up the internal driver so this protocol actually functions.
         This method will be called before the object is fully constructed
         !
131    * @param bufferLength
132    * @param arrivalHandler
133    */
134   protected void setupDriver(int bufferLength) {
135     //get a new rx-driver in non-destructive mode (ie: blocking writes
          on full buffer)
136     //This will return null if this buffer is in a shadow machine.
137     _drv = owner.getMachineDriver().getTPIFDriverRx(getExternalID(), new
           Runnable() {
138       @Override
139       public void run() {
140         registerEvent(new ReturnableEnvelope<Q, R>(_drv,
              RemoteProcedureCall.this));
141       }
142     }, false, bufferLength);
143     _drv.accept(); //accept the first item if this is a buffered driver.
144   }
145
146   /**
147    * Gets a connector that can be used to query this remote procedure
         call.
148    * @param replyHandler the event handler that will receive replies
         from queries sent to the corresponding RPC.
149    * @return a new open connector object.
150    */
151   public RemoteProcedureCallConnector<Q, R> newConnector(Handler<
        Envelope<R>> replyHandler) {
152     return new RemoteProcedureCallConnector<Q, R>(this, replyHandler);
153   }
154
155   /**
156    * As with newConnector but this connector will automatically close
         after one query.
157    * @param replyHandler
158    * @return
159    */
160   public RemoteProcedureCallConnector<Q, R> newEphemeralConnector(
```

```
        Handler<Envelope<R>> replyHandler) {
161     return new RemoteProcedureCallConnector<Q, R>(this, replyHandler).
          setEphemeral(true);
162   }
163 
164   /**
165    * Queries the remote machine and waits for the response. This is not
         a good design in general! (Too many opportunities for deadlock or
          memory exhaustion)
166    *
167    * @param query
168    * @return
169    */
170   public R blockingQuery(Q query) {
171     RemoteProcedureCallConnector<Q, R> connector = newConnector(null);
172     R response = connector.sendAndWait(query);
173     try {
174       connector.shutdown();
175     } catch (IOException e) {
176       e.printStackTrace();
177     }
178     return response;
179   }
180 
181   /**
182    * Issues a new non-blocking query on an ephemeral connector.
183    * @param query
184    * @param replyHandler
185    */
186   public void query(Q query, Handler<Envelope<R>> replyHandler) {
187     (new RemoteProcedureCallConnector<Q, R>(this, replyHandler).
          setEphemeral(true)).query(query);
188   }
189 }
```

A.2 Period.java

```java
package mjava.core.time;

import mjava.core.Handler;
import mjava.core.runtime.Platform;
import mjava.core.runtime.drivers.AlarmDriver;

/**
 * Provides an interval timer for mjava machines.
 *
 * This event source will issue an event approximately every period in
     milliseconds, as long as the event handler is executed before the
     next period is due.
 *
 * @author gary
 */
public class Period extends TimeEventSource implements Runnable {

  protected AlarmDriver drv = Platform.getPlatform().locateMachineDriver
      ().getAlarmDriver(this);

  protected final long periodMS;
  protected long lastScheduledFor;

  public Period(long periodMS, final Handler<TimeEvent> handler) {
    //first give the event source a null handler because we need to use
        a proxy handler to detect when the event handler has finished
        executing.
    super(null);

    this.periodMS = periodMS;
    //assume we were most recently activated right now.
    this.lastScheduledFor = System.currentTimeMillis();

    setHandler(new Handler<TimeEvent>() {
      @Override
      public void handle(TimeEvent info) {
        //run client code:
        handler.handle(info);

        //schedule the next period event as long as the periodic wasn't
            cancelled in the event handler.
        if (!isCancelled) again();
      }
    });

    //Startup:
    again();
```

```
42    }
43
44    /**
45     * Called after the event handler has executed to indicate we should
          schedule again.
46     *
47     * Clients of the Periodic Timed Event Source do not call again().
48     */
49    protected void again() {
50      this.lastScheduledFor = lastScheduledFor+periodMS;
51      drv.setAlarm(lastScheduledFor);
52    }
53
54    boolean isCancelled = false;
55
56    @Override
57    public void cancel() {
58      isCancelled = true;
59      drv.cancel();
60    }
61
62    @Override
63    public void run() {
64      doHandle(new TimeEvent(this));
65    }
66
67    /**
68     * Periods cannot be 'agained'
69     */
70    @Override
71    public void again(long newValue) {
72      throw new UnsupportedOperationException();
73    }
74
75  }
```

Appendix B

Water Tank Level Control Application

Contents

A.1 RemoteProcedureCall.java 320

A.2 Period.java 325

Example Machine Java code for the water tank level control application (shown in figure 3.2) is reproduced here. Note that this is just a structural example: the sensors use random numbers as their sources, the actuators just print messages to the console, and an unrealistically simple pump control calculation is used by the `FlowController`.

B.1 FlowController

```
1  package examples.watertank;
2
3  import mjava.core.Handler;
4  import mjava.core.Start;
5  import mjava.core.time.Delay;
6  import mjava.core.time.TimeEvent;
7  import mjava.core.tpif.Envelope;
8  import mjava.core.tpif.OverwritingBuffer;
9  import mjava.tools.chi.Experiment;
10
11 public class FlowController extends Start {
12
13   private FlowController() {}
14
15   public static void main(String[] args) {
```

```
16    start(FlowController.class);
17  }
18
19  private int calculateRequiredPumpPower(int tankLevel) {
20    return 100-tankLevel;
21  }
22
23  private RefillPump pump;
24
25  public final OverwritingBuffer<Integer> tankLevel = new
      OverwritingBuffer<Integer>(1, new Handler<Envelope<Integer>>() {
26    @Override
27    public void handle(Envelope<Integer> info) {
28      int tankLevelNow = info.getPayload();
29      pump.power.send(calculateRequiredPumpPower(tankLevelNow));
30    }
31  });
32
33  @Override
34  protected void internal() {
35
36    newMachine(RefillPump.class, new Handler<RefillPump>() {
37
38      @Override
39      public void handle(RefillPump info) {
40        pump = info;
41
42        //Got the pump so can now create our sensors:
43        setupMachine(LevelSensor.class, FlowController.this, null);
44      }
45    });
46
47
48    new Delay(20*1000, new Handler<TimeEvent>() {
49      @Override
50      public void handle(TimeEvent info) {
51        Experiment.terminate();
52      }
53    });
54
55  }
56
57 }
```

B.2 RefillPump

```
1 package examples.watertank;
2
3 import mjava.core.Handler;
```

```
import mjava.core.Machine;
import mjava.core.tpif.Envelope;
import mjava.core.tpif.OverwritingBuffer;

public class RefillPump extends Machine {

  private RefillPump() {}

  public final OverwritingBuffer<Integer> power = new OverwritingBuffer<
      Integer>(new Handler<Envelope<Integer>>() {

    @Override
    public void handle(Envelope<Integer> info) {
      setPower(info.getPayload());
    }
  });

  private void setPower(int toPower) {
    log("setting pump power to "+toPower+"%");
  }

}
```

B.3 LevelSensor

```
package examples.watertank;

import mjava.core.EventCombiner;
import mjava.core.Handler;
import mjava.core.Pair;
import mjava.core.SetupableMachine;
import mjava.core.time.Period;
import mjava.core.time.TimeEvent;
import mjava.core.tpif.Envelope;

public class LevelSensor extends SetupableMachine<FlowController> {

  private LevelSensor() {}

  public static final int REPORT_MILLIS = 100;
  public static final int POLL_MILLIS = 50;

  protected static final int EMERGENCY_THRESHOLD = 90;

  private int currentTankLevel;

  private UnreliableSensor sensorA;
  private UnreliableSensor sensorB;

```

```
25  private EmergencyValve emergency;
26
27  //A combiner that fires when both sensors are ready and when the timed
        interval has happened.
28  private EventCombiner<TimeEvent, Pair<Envelope<Integer>, Envelope<
      Integer>>> timedPollCombiner = new EventCombiner<TimeEvent, Pair<
      Envelope<Integer>,Envelope<Integer>>>(new Handler<Pair<TimeEvent,
      Pair<Envelope<Integer>,Envelope<Integer>>>>() {
29    @Override
30    public void handle(Pair<TimeEvent, Pair<Envelope<Integer>, Envelope<
          Integer>>> info) {
31
32      int sensorAValue = info.right.left.getPayload();
33      int sensorBValue = info.right.right.getPayload();
34
35      currentTankLevel = (sensorAValue+sensorBValue)/2;
36
37      if (currentTankLevel>EMERGENCY_THRESHOLD) emergency.openValve.
          signal();
38
39      //query the sensors again:
40      querySensors();
41
42    }
43  });
44
45  //An event combiner so we only execute once we've received the latest
      from each sensor:
46  private EventCombiner<Envelope<Integer>, Envelope<Integer>>
      sensorResponseCombiner = new EventCombiner<Envelope<Integer>,
      Envelope<Integer>>(timedPollCombiner.getRightHandler());
47
48  @Override
49  protected void internal() {
50
51    //Get sensorA
52    newMachine(UnreliableSensor.class, new Handler<UnreliableSensor>() {
53      @Override
54      public void handle(UnreliableSensor info) {
55        sensorA = info;
56        sensorA.currentLevel.fetch(sensorResponseCombiner.getLeftHandler
            ());
57      }
58    });
59
60    //Get sensorB
61    newMachine(UnreliableSensor.class, new Handler<UnreliableSensor>() {
62      @Override
63      public void handle(UnreliableSensor info) {
```

```
64          sensorB = info;
65          sensorB.currentLevel.fetch(sensorResponseCombiner.
                getRightHandler());
66        }
67      });
68
69      newMachine(EmergencyValve.class, new Handler<EmergencyValve>() {
70        @Override
71        public void handle(EmergencyValve info) {
72          emergency = info;
73        }
74      });
75
76      //A periodic task to poll the sensors: (event combined with sensor
            responses)
77      new Period(POLL_MILLIS, timedPollCombiner.getLeftHandler());
78    }
79
80    private void querySensors() {
81      if (sensorA!=null) sensorA.currentLevel.fetch(sensorResponseCombiner
            .getLeftHandler());
82      if (sensorB!=null) sensorB.currentLevel.fetch(sensorResponseCombiner
            .getRightHandler());
83    }
84
85    @Override
86    protected void setup(final FlowController controller) {
87
88      //Now we know who the controller is a periodic task can be setup to
            send it tank level data:
89      new Period(REPORT_MILLIS, new Handler<TimeEvent>() {
90
91        @Override
92        public void handle(TimeEvent info) {
93          controller.tankLevel.send(currentTankLevel);
94        }
95      });
96
97    }
98
99  }
```

B.4 EmergencyValve

```
1 package examples.watertank;
2
3 import mjava.core.Handler;
4 import mjava.core.Machine;
5 import mjava.core.Nothing;
```

```
6 import mjava.core.tpif.Envelope;
7 import mjava.core.tpif.Stimulus;
8
9 public class EmergencyValve extends Machine {
10
11   private EmergencyValve() {}
12
13   public final Stimulus openValve = new Stimulus(new Handler<Envelope<
         Nothing>>() {
14     @Override
15     public void handle(Envelope<Nothing> info) {
16       //Payload must always be read!
17       info.getPayload();
18       log("EMERGENCY!");
19     }
20   });
21
22 }
```

B.5 UnreliableSensor

```
1 package examples.watertank;
2
3 import java.util.Random;
4
5 import mjava.core.Handler;
6 import mjava.core.Machine;
7 import mjava.core.Nothing;
8 import mjava.core.tpif.RemoteDataFetch;
9 import mjava.core.tpif.ReturnableEnvelope;
10
11 public class UnreliableSensor extends Machine {
12
13   private UnreliableSensor() {}
14
15   private Random rng = new Random();
16
17   public final RemoteDataFetch<Integer> currentLevel = new
         RemoteDataFetch<Integer>(new Handler<ReturnableEnvelope<Nothing,
         Integer>>() {
18     @Override
19     public void handle(ReturnableEnvelope<Nothing, Integer> info) {
20       //let's say (50+-45)%
21       info.reply(5+rng.nextInt(90));
22     }
23   });
24
25 }
```

Appendix C

Microbenchmark Source Code

Contents

B.1 FlowController . . . 327
B.2 RefillPump . . . 328
B.3 LevelSensor . . . 329
B.4 EmergencyValve . . . 331
B.5 UnreliableSensor . . . 332

C.1 SpeedTest

C.1.1 SpeedTest.java

```
1  package examples.speedtest;
2
3  import mjava.core.Handler;
4  import mjava.core.Pair;
5  import mjava.core.Start;
6  import mjava.core.time.Delay;
7  import mjava.core.time.Period;
8  import mjava.core.time.TimeEvent;
9  import mjava.core.tpif.BoundedBuffer;
10 import mjava.core.tpif.Envelope;
11 import mjava.tools.chi.Experiment;
12 import mjava.tools.chi.runtime.Network;
13
14 /**
15  * A speed/latency tester for mJava.
16  * Will add a new machine pair every ADD_INTERVAL_MS seconds until the
       timeout
17  *
```

```
18  * Will print a message on every pair addition with the current
      throughput/latency figures.
19  *
20  * @author gary
21  */
22 public class SpeedTest extends Start {
23
24   public static final int DEATH_COUNTER = 5;
25
26   public static final int TERMINATE_MS = 1000*60*10; //10mins
27
28   /**
29    * Interval to add new message exchanging machines at:
30    */
31   public static final int ADD_INTERVAL_MS = 10_000;
32
33   /**
34    * Message size exchanged in bytes
35    */
36   public static final int MSG_SIZE = 5;
37
38
39   private SpeedTest() {}
40
41   private int pairCount = 0;
42
43   ///during the last interval:
44   //bytes
45   private long bytesTransferred;
46   //n
47   private long messagesExchanged;
48   //ns
49   private long latencyEncountered;
50   //ms
51   private long lastReported;
52
53   //totals:
54   private long messagesEver;
55   private long transferredEver;
56
57   public static void main(String[] args) {
58     start(SpeedTest.class);
59   }
60
61   @Override
62   protected void internal() {
63
64     //Add a new communicating pair every
65     new Period(ADD_INTERVAL_MS, new Handler<TimeEvent>() {
```

```
66          @Override
67          public void handle(TimeEvent info) {
68            addPair();
69          }
70        });
71
72        new Delay(TERMINATE_MS, true, new Handler<TimeEvent>() {
73
74          @Override
75          public void handle(TimeEvent info) {
76            log("Terminating due to timeout of (ms): "+TERMINATE_MS);
77            Experiment.terminate();
78          }
79        });
80
81        log("#i#SpeedTest#, processors, pairs, avgLatencyuS, rateB/s,
              messages");
82        log("#i#SpeedTest# Message size: "+MSG_SIZE);
83
84        addPair();
85
86        //Experiment.waitDumpExit();
87
88      }
89
90      /**
91       * Adds a new pair of communicating machines
92       */
93      private void addPair() {
94        report();
95
96        //Create both the ping-client and server here and pass the server
              reference to the client.
97        PingClient newPingClient = newMachine(PingClient.class);
98        PingServer newPingSrv = newMachine(PingServer.class);
99
100       //Send the new ping client our reference (for reporting) and the
              server that it should ping.
101       newPingClient.setup.send(Pair.p(this, newPingSrv));
102
103       pairCount++;
104       log("New PingClient ("+newPingClient+") and PingServer created!");
105     }
106
107
108     /**
109      * A pair will periodically report back some total latency and message
              count info.
110      */
```

```
111 public final BoundedBuffer<PingClientReport> reportExchange = new
        BoundedBuffer<PingClientReport>(50, new Handler<Envelope<
        PingClientReport>>() {
112
113   @Override
114   public void handle(Envelope<PingClientReport> info) {
115     long latency = info.getPayload().latencyNanos;
116     int messages = info.getPayload().messageCount;
117
118     messagesEver+=messages;
119     transferredEver += MSG_SIZE*messages;
120
121     bytesTransferred += MSG_SIZE*messages;
122     messagesExchanged+=messages;
123     latencyEncountered += latency;
124   }
125
126 });
127
128 private int deadFor = 0;
129 private final int pCount = Network.width()*Network.height();
130 private final String datalogPrefix = "#d#SpeedTest#, "+pCount+", ";
131
132 private void report() {
133   long now = System.currentTimeMillis();
134   long interval = now-lastReported;
135
136   if (messagesExchanged>0) {
137     long avgLatencyuS = (latencyEncountered/messagesExchanged)/1000;
138     long rate = (bytesTransferred*1000l)/interval;
139
140     log(datalogPrefix+pairCount+", "+avgLatencyuS+", "+rate+", "+
          messagesExchanged);
141     deadFor = 0;
142   } else {
143     log("Pairs: "+pairCount+". No messages exchanged in last interval.
          ");
144     deadFor++;
145     //quit if we've been broken for too long:
146     if (deadFor>DEATH_COUNTER) {
147       log("We've apparently died. Terminating.");
148       Experiment.terminate();
149     }
150   }
151
152   messagesExchanged = 0;
153   latencyEncountered = 0;
154   bytesTransferred = 0;
155
```

```
156     lastReported = now;
157   }
158
159 }
```

C.1.2 PingClient.java

```
1  package examples.speedtest;
2
3  import java.util.Collections;
4
5  import examples.speedtest.PingServer.PingMsg;
6  import mjava.core.EventCombiner;
7  import mjava.core.Handler;
8  import mjava.core.Machine;
9  import mjava.core.Nothing;
10 import mjava.core.Pair;
11 import mjava.core.VisibleMachineDependencies;
12 import mjava.core.time.Period;
13 import mjava.core.time.TimeEvent;
14 import mjava.core.tpif.BoundedBuffer;
15 import mjava.core.tpif.Envelope;
16 import mjava.immutable.ImmutableSet;
17
18 public class PingClient extends Machine implements
       VisibleMachineDependencies{
19
20   private PingClient() {}
21
22   private long lastSentNanos;
23
24   private SpeedTest host;
25
26   public final BoundedBuffer<Pair<SpeedTest, PingServer>> setup = new
         BoundedBuffer<Pair<SpeedTest, PingServer>>(1, new Handler<Envelope
         <Pair<SpeedTest, PingServer>>>() {
27     @Override
28     public void handle(Envelope<Pair<SpeedTest, PingServer>> info) {
29       host = info.getPayload().left;
30       srv = info.getPayload().right;
31
32       //now we can start:
33       lastSentNanos = System.nanoTime();
34       srv.replyService.query(new PingMsg(), new PingReplyHandler());
35       //con.query(new PingMsg());
36     }
37   });
38
39   private PingServer srv;
40
```

```
41  private long latency;
42  private int messages;
43
44
45  private final class PingReplyHandler implements Handler<Envelope<
        PingMsg>> {
46    @Override
47    public void handle(Envelope<PingMsg> info) {
48      //don't actually care about the reply, but it's imperative to read
            it anyway!
49      info.getPayload();
50
51      //Woop replied!
52      long latencyNanos = System.nanoTime()-lastSentNanos;
53
54      latency+=latencyNanos;
55      messages++;
56
57
58      lastSentNanos = System.nanoTime();
59      srv.replyService.query(new PingMsg(), PingReplyHandler.this);
60
61    }
62  }
63
64
65  public static final int REPORT_INTERVAL_MS = 2000;
66
67  private EventCombiner<TimeEvent, Nothing> combiner = new EventCombiner
        <TimeEvent, Nothing>(new Handler<Pair<TimeEvent, Nothing>>() {
68
69    @Override
70    public void handle(Pair<TimeEvent, Nothing> info) {
71      if (host!=null) {
72        host.reportExchange.send(new PingClientReport(messages, latency)
              , combiner.getRightHandler());
73        latency = 0;
74        messages = 0;
75      } else {
76        combiner.rightSideReady();
77      }
78    }
79  });
80
81  @Override
82  protected void internal() {
83    lastSentNanos = System.nanoTime();
84
85    //Setup a periodic task to report the current latency and message
```

```
      count to the host:
86    new Period(REPORT_INTERVAL_MS, combiner.getLeftHandler());
87
88    combiner.rightSideReady();
89  }
90
91  @Override
92  public ImmutableSet<Machine> _getReferencedMachines() {
93    return new ImmutableSet<>(Collections.singleton((Machine)srv));
94  }
95
96 }
```

C.1.3 PingServer.java

```
1  package examples.speedtest;
2
3  import java.io.DataInputStream;
4  import java.io.DataOutputStream;
5  import java.io.IOException;
6
7  import mjava.core.Handler;
8  import mjava.core.Immutable;
9  import mjava.core.Machine;
10 import mjava.core.tpif.RemoteProcedureCall;
11 import mjava.core.tpif.ReturnableEnvelope;
12
13 public class PingServer extends Machine {
14
15   /**
16    * This is the message class for the speed tester.
17    * This doesn't actually hold a message but is a data generator and
         sink.
18    * @author gary
19    */
20   public static class PingMsg implements Immutable {
21
22     public PingMsg() {
23     }
24
25     public PingMsg(DataInputStream in) throws IOException {
26       for (int i=0; i<SpeedTest.MSG_SIZE; i++) {
27         in.read();
28       }
29     }
30
31     @Override
32     public void flatten(DataOutputStream out) throws IOException {
33       for (int i=0; i<SpeedTest.MSG_SIZE; i++) {
34         out.write((byte)i);
```

```
35          }
36        }
37
38      }
39
40      private PingServer() {}
41
42      /**
43       * Just replies to the client immediately.
44       */
45      public final RemoteProcedureCall<PingMsg, PingMsg> replyService = new
            RemoteProcedureCall<>(new Handler<ReturnableEnvelope<PingMsg,
            PingMsg>>() {
46        @Override
47        public void handle(ReturnableEnvelope<PingMsg, PingMsg> info) {
48          info.reply(info.getPayload());
49        }
50      });
51
52      protected void internal() {};
53
54    }
```

C.1.4 PingClientReport.java

```
1   package examples.speedtest;
2
3   import java.io.DataInputStream;
4   import java.io.DataOutputStream;
5   import java.io.IOException;
6
7   import mjava.core.Immutable;
8
9   public class PingClientReport implements Immutable {
10
11    public final int messageCount;
12    public final long latencyNanos;
13
14    @Override
15    public void flatten(DataOutputStream out) throws IOException {
16      out.writeInt(messageCount);
17      out.writeLong(latencyNanos);
18    }
19
20    public PingClientReport(int messageCount, long latencyNanos) {
21      this.messageCount = messageCount;
22      this.latencyNanos = latencyNanos;
23    }
24
25    public PingClientReport(DataInputStream in) throws IOException {
```

```
26      messageCount = in.readInt();
27      latencyNanos = in.readLong();
28    }
29  }
```

C.2 SpeedTestHP

C.2.1 PingClientHP.java

```
1  package examples.speedtest.highperf;
2
3  import examples.speedtest.highperf.PingServerHP.PingMsg;
4  import mjava.core.EventCombiner;
5  import mjava.core.Handler;
6  import mjava.core.Machine;
7  import mjava.core.Nothing;
8  import mjava.core.Pair;
9  import mjava.core.time.Period;
10 import mjava.core.time.TimeEvent;
11 import mjava.core.tpif.Envelope;
12 import mjava.core.tpif.OverwritingBuffer.OverwritingBufferConnector;
13 import mjava.core.tpif.Signal;
14
15 public class PingClientHP extends Machine {
16
17   private PingClientHP() {}
18
19   private long lastSentNanos;
20
21   private SpeedTestHP host;
22
23   public final Signal<Pair<SpeedTestHP, PingServerHP>> setup = new
         Signal<Pair<SpeedTestHP, PingServerHP>>(new Handler<Envelope<Pair<
         SpeedTestHP, PingServerHP>>>() {
24     @Override
25     public void handle(Envelope<Pair<SpeedTestHP, PingServerHP>> info) {
26       host = info.getPayload().left;
27       toSrv = info.getPayload().right.replyService.newConnector();
28
29       //now we can start:
30       lastSentNanos = System.nanoTime();
31       toSrv.send(new PingMsg());
32     }
33   });
34
35
36   private OverwritingBufferConnector<PingMsg> toSrv;
37
38   public final Signal<PingMsg> returnChannel = new Signal<>(new
```

```
        PingReplyHandler());
39
40    private long latency;
41    private int messages;
42
43
44    private final class PingReplyHandler implements Handler<Envelope<
        PingMsg>> {
45      @Override
46      public void handle(Envelope<PingMsg> info) {
47        //don't actually care about the reply, but it's imperative to read
            it anyway!
48        info.getPayload();
49
50        //Woop replied!
51        long latencyNanos = System.nanoTime()-lastSentNanos;
52
53        latency+=latencyNanos;
54        messages++;
55
56
57        lastSentNanos = System.nanoTime();
58        toSrv.send(info.getPayload());
59      }
60    }
61
62
63    public static final int REPORT_INTERVAL_MS = 2000;
64
65    private EventCombiner<TimeEvent, Nothing> combiner = new EventCombiner
        <TimeEvent, Nothing>(new Handler<Pair<TimeEvent, Nothing>>() {
66
67      @Override
68      public void handle(Pair<TimeEvent, Nothing> info) {
69        if (host!=null) {
70          host.reportExchange.send(new PingClientReport(messages, latency)
              , combiner.getRightHandler());
71          latency = 0;
72          messages = 0;
73        } else {
74          combiner.rightSideReady();
75        }
76      }
77    });
78
79    @Override
80    protected void internal() {
81      //log("pingclient internal(): 1");
82
```

```
83      lastSentNanos = System.nanoTime();
84
85      //Setup a periodic task to report the current latency and message
           count to the host:
86      new Period(REPORT_INTERVAL_MS, combiner.getLeftHandler());
87
88      combiner.rightSideReady();
89
90      //log("pingclient internal(): 2");
91    }
92
93  }
```

C.2.2 PingServerHP.java

```
1  package examples.speedtest.highperf;
2
3  import java.io.DataInputStream;
4  import java.io.DataOutputStream;
5  import java.io.IOException;
6
7  import mjava.core.Handler;
8  import mjava.core.Immutable;
9  import mjava.core.SetupableMachine;
10 import mjava.core.tpif.Envelope;
11 import mjava.core.tpif.OverwritingBuffer.OverwritingBufferConnector;
12 import mjava.core.tpif.Signal;
13 import mjava.tools.chi.Experiment;
14
15 public class PingServerHP extends SetupableMachine<PingClientHP> {
16
17
18   /**
19    * This is the message class for the speed tester.
20    * This doesn't actually hold a message but is a data generator and
          sink.
21    * @author gary
22    */
23   public static class PingMsg implements Immutable {
24
25     //use an experimental parameter:
26     private final int MSG_SIZE = Experiment.getExpParamAsInt(0);
27
28     public PingMsg() {
29     }
30
31     public PingMsg(DataInputStream in) throws IOException {
32       for (int i=0; i<MSG_SIZE; i++) {
33         in.read();
34       }
```

```
35     }
36
37     @Override
38     public void flatten(DataOutputStream out) throws IOException {
39       for (int i=0; i<MSG_SIZE; i++) {
40         out.write((byte)i);
41       }
42     }
43
44   }
45
46   private PingServerHP() {}
47
48   /**
49    * Just replies to the client immediately using the predefined
         connector.
50    */
51   public final Signal<PingMsg> replyService = new Signal<>(new Handler<
       Envelope<PingMsg>>(){
52     @Override
53     public void handle(Envelope<PingMsg> info) {
54       toClient.send(info.getPayload());
55     }
56   });
57
58   private OverwritingBufferConnector<PingMsg> toClient;
59
60   @Override
61   protected void setup(PingClientHP value) {
62     toClient = value.returnChannel.newConnector();
63   };
64
65 }
```

C.3 DistributedMD5

C.3.1 DistributedMD5.java

```
1  package examples.compute;
2
3  import examples.speedtest.SpeedTest;
4  import mjava.core.Handler;
5  import mjava.core.Start;
6  import mjava.core.time.Delay;
7  import mjava.core.time.Period;
8  import mjava.core.time.TimeEvent;
9  import mjava.core.tpif.BoundedBuffer;
10 import mjava.core.tpif.Envelope;
11 import mjava.tools.chi.Experiment;
```

```
12 import mjava.tools.chi.runtime.Network;
13
14 public class DistributedMD5 extends Start {
15
16   public static final int TERMINATE_MS = SpeedTest.TERMINATE_MS; //10
       mins
17
18   /**
19    * Interval to add new message exchanging machines at:
20    */
21   public static final int ADD_INTERVAL_MS = SpeedTest.ADD_INTERVAL_MS;
22
23   private DistributedMD5() {}
24
25   private int workerCount = 0;
26
27   ///during the last interval:
28   //hashes
29   private long hashCount;
30   //ms
31   private long lastReported;
32
33   //totals:
34   @SuppressWarnings("unused")
35   private long hashesEver;
36
37   public static void main(String[] args) {
38     start(DistributedMD5.class);
39   }
40
41   @Override
42   protected void internal() {
43
44     //Add a new communicating pair every
45     new Period(ADD_INTERVAL_MS, new Handler<TimeEvent>() {
46       @Override
47       public void handle(TimeEvent info) {
48         addWorker();
49       }
50     });
51
52     new Delay(TERMINATE_MS, true, new Handler<TimeEvent>() {
53
54       @Override
55       public void handle(TimeEvent info) {
56         log("Terminating due to timeout of (ms): "+TERMINATE_MS);
57         Experiment.terminate();
58       }
59     });
```

```
60
61      log("#i#DistributedMD5#, processors, workers, hashes/s");
62      log("#i#DistributedMD5#, batch size: "+MD5Worker.BATCH_SIZE);
63
64      addWorker();
65
66    }
67
68    /**
69     * Adds a new pair of communicating machines
70     */
71    private void addWorker() {
72      report();
73      setupMachine(MD5Worker.class, this, new Handler<MD5Worker>() {
74
75        @Override
76        public void handle(MD5Worker info) {
77          workerCount++;
78          log("New MD5Worker ("+info+") created!");
79        }
80      });
81    }
82
83
84    /**
85     * periodically workers will report how much they have completed.
86     */
87    public final BoundedBuffer<Integer> reportHashesComplete = new
         BoundedBuffer<Integer>(50, new Handler<Envelope<Integer>>() {
88
89      @Override
90      public void handle(Envelope<Integer> info) {
91        long completed = info.getPayload();
92
93        hashCount += completed;
94        hashesEver += completed;
95
96      }
97
98    });
99
100   private int deadFor = 0;
101   private final int pCount = Network.width()*Network.height();
102   private final String datalogPrefix = "#d#DistributedMD5#, "+pCount+",
         ";
103
104   private void report() {
105     long now = System.currentTimeMillis();
106     long interval = now-lastReported;
```

```
107
108     if (hashCount>0) {
109       long rate = (hashCount*1000l)/interval;
110       //int krate = (int) (rate/1000);
111
112       log(datalogPrefix+workerCount+", "+rate);
113       deadFor = 0;
114     } else {
115       log("Workers: "+workerCount+". No work done in last interval.");
116       deadFor++;
117       //quit if we've been broken for too long:
118       if (deadFor>SpeedTest.DEATH_COUNTER) {
119         log("We've apparently died. Terminating.");
120         Experiment.terminate();
121       }
122     }
123
124     hashCount = 0;
125
126     lastReported = now;
127   }
128
129 }
```

C.3.2 MD5Worker.java

```
1  package examples.compute;
2
3  import examples.speedtest.PingClient;
4  import mjava.core.EventCombiner;
5  import mjava.core.Handler;
6  import mjava.core.Nothing;
7  import mjava.core.Pair;
8  import mjava.core.SetupableMachine;
9  import mjava.core.time.Period;
10 import mjava.core.time.TimeEvent;
11 import mjava.core.time.Yield;
12 import mjava.tools.chi.tests.network.MD5;
13
14 /**
15  * The MD5 worker just calculates MD5 digests continuously.
16  *
17  * it will periodically report back to the manager.
18  *
19  * @author gary
20  *
21  */
22 public class MD5Worker extends SetupableMachine<DistributedMD5> {
23
24   private MD5Worker() {}
```

```
25
26    private DistributedMD5 manager;
27
28    private int hashesSinceLastReport = 0;
29
30    public static final int REPORT_INTERVAL_MS = PingClient.
          REPORT_INTERVAL_MS;
31
32    /**
33     * The number of MD5s to do in a batch before yielding to other events
34     */
35    public static final int BATCH_SIZE = 1;
36
37
38    private EventCombiner<TimeEvent, Nothing> combiner = new EventCombiner
          <TimeEvent, Nothing>(new Handler<Pair<TimeEvent, Nothing>>() {
39
40      @Override
41      public void handle(Pair<TimeEvent, Nothing> info) {
42        if (manager!=null) {
43          manager.reportHashesComplete.send(hashesSinceLastReport,
              combiner.getRightHandler());
44          hashesSinceLastReport = 0;
45        } else {
46          combiner.rightSideReady();
47        }
48      }
49    });
50
51    @Override
52    protected void internal() {
53      //1) setup reporting task:
54
55      new Period(REPORT_INTERVAL_MS, combiner.getLeftHandler());
56
57      //We can output to the manager already:
58      combiner.rightSideReady();
59
60    }
61
62    /**
63     * does the MD5 work.
64     */
65    @SuppressWarnings("unused") //It's not unused!
66    private Yield md5er = new Yield(new Handler<Nothing>() {
67
68      @Override
69      public void handle(Nothing info) {
70
```

```
71          for (int i=0;i<BATCH_SIZE;i++) {
72            increment(input);
73            getMD5(input);
74            hashesSinceLastReport++;
75          }
76
77          //go again later
78          md5er.yeild();
79        }
80      });
81
82      /**
83       * the value we're currently hashing.
84       */
85      private byte[] input = new byte[64];
86
87      private MD5 m = new MD5();
88      private byte[] getMD5(byte[] input) {
89        m.reset();
90        m.update(input);
91        return m.getHash();
92      }
93
94      public static void increment(byte[] largeNumber) {
95        int onIdx = largeNumber.length-1;
96
97        while (onIdx>=0) {
98          largeNumber[onIdx]++;
99          if (largeNumber[onIdx]==0) {
100           onIdx--;
101         } else {
102           return;
103         }
104       }
105     }
106
107     @Override
108     protected void setup(DistributedMD5 value) {
109       manager = value;
110     }
111
112   }
```

C.4 Dining Philosophers

C.4.1 Fork.java

```
1 package examples.philosophers;
2
```

```
3 import mjava.core.Handler;
4 import mjava.core.Machine;
5 import mjava.core.Nothing;
6 import mjava.core.tpif.BoundedBuffer;
7 import mjava.core.tpif.Envelope;
8 import mjava.core.tpif.RemoteProcedureCall;
9 import mjava.core.tpif.ReturnableEnvelope;
10
11 /**
12  * A 'fork' in the dining philosophers problem.
13  *
14  * Contains an RPC that returns true if the fork was gained, false if
       the request was denied.
15  * May block arbitrarily long; never guaranteed to return false.
16  *
17  * @author gary
18  */
19 public class Fork extends Machine {
20
21   private Fork() {}
22
23   protected Philosopher owner;
24
25   public final RemoteProcedureCall<Nothing, Boolean> gainFork = new
        RemoteProcedureCall<Nothing, Boolean>(1, new Handler<
        ReturnableEnvelope<Nothing, Boolean>>() {
26     @Override
27     public void handle(ReturnableEnvelope<Nothing, Boolean> info) {
28       if (owner==null) {
29         owner = (Philosopher) info.getRequestingMachine();
30         info.reply(true);
31         //log("now owned by: "+owner);
32       } else {
33         info.reply(false);
34         //log("failed request to get fork by: "+info.
              getRequestingMachine()+", it's owned by: "+owner);
35       }
36     }
37   });
38
39   public final BoundedBuffer<Philosopher> releaseFork = new
        BoundedBuffer<Philosopher>(1, new Handler<Envelope<Philosopher>>()
         {
40     @Override
41     public void handle(Envelope<Philosopher> info) {
42       if (!owner.equals(info.getPayload())) {
43         throw new ForkException("A philosopher tried to release a fork
              they didn't own.");
44       }
```

```
45        //log("fork released by "+owner+" and now free.");
46        owner = null;
47      }
48    });
49
50    @Override
51    protected void internal() {
52      //Forks have high priority compared to philosophers and the release
          () has higher priority than gain()
53      setPriority(5);
54      releaseFork.setPriority(5);
55    }
56
57  }
```

C.4.2 Dining.java

```
1  package examples.philosophers;
2
3  import examples.philosophers.Philosopher.ProvideFork;
4  import mjava.core.Start;
5  import mjava.core.tpif.BoundedBuffer;
6  import mjava.tools.chi.Experiment;
7  import mjava.core.tpif.Envelope;
8  import mjava.core.Handler;
9
10 /**
11  * A dining philosophers implementation in Machine Java.
12  * Both philosophers and forks are represented by machines.
13  * @author gary
14  */
15 public class Dining extends Start {
16
17   public static final int PHILOSOPHER_COUNT = 100;
18
19   public static void main(String[] args) {
20     start(Dining.class);
21   }
22
23   protected Dining() {}
24
25   @Override
26   protected void internal() {
27     log("Dining.internal()");
28     //Request and provide the first philosopher with no forks... She can
           make her own.
29     setupMachine(Philosopher.class, new ProvideFork(this,
          PHILOSOPHER_COUNT-1, null, null), null);
30
31     //Don't execute this experiment for ever:
```

```
32      Experiment.waitDumpExit();
33    }
34
35    private boolean loggingEnabled = true;
36
37    public final BoundedBuffer<String> eatingNotification = new
        BoundedBuffer<String>(100, new Handler<Envelope<String>>() {
38      @Override
39      public void handle(Envelope<String> info) {
40        if (loggingEnabled) log("@"+System.currentTimeMillis()/1000+"s: "+
            info.getPayload());
41      }
42    });
43
44
45  }
```

C.4.3 Philosopher.java

```
1  package examples.philosophers;
2
3  import java.io.DataInputStream;
4  import java.io.DataOutputStream;
5  import java.io.IOException;
6  import java.util.Arrays;
7  import java.util.HashSet;
8  import java.util.Random;
9
10 import mjava.core.Handler;
11 import mjava.core.Immutable;
12 import mjava.core.Machine;
13 import mjava.core.Nothing;
14 import mjava.core.SetupableMachine;
15 import mjava.core.VisibleDescription;
16 import mjava.core.VisibleMachineDependencies;
17 import mjava.core.time.Delay;
18 import mjava.core.time.TimeEvent;
19 import mjava.core.tpif.Envelope;
20 import mjava.immutable.ImmutableSet;
21 import mjava.immutable.NotInflatableException;
22 import mjava.immutable.Utilities;
23
24 /**
25  * A hungry philosopher in the Dining Philosophers problem.
26  * @author gary
27  *
28  */
29 public class Philosopher extends SetupableMachine<examples.philosophers.
     Philosopher.ProvideFork> implements VisibleMachineDependencies,
     VisibleDescription {
```

```
30
31   @Override
32   public String _getDescription() {
33     return Integer.toString(Dining.PHILOSOPHER_COUNT-ourID);
34   }
35
36   @Override
37   public ImmutableSet<Machine> _getReferencedMachines() {
38     return new ImmutableSet<>(new HashSet<>(Arrays.asList((Machine)
         firstFork,(Machine)secondFork)));
39   }
40
41   //Higher priority fork
42   private Fork firstFork;
43   //Lower priority fork
44   private Fork secondFork;
45
46   //Let's keep a simple ID around for debugging/inspection
47   private int ourID;
48
49   private Philosopher(){
50     //ENSURE_SINGLETON();
51   }
52
53   @Override
54   protected void internal() {
55     //Only make an RNG if this is a real machine:
56     rng = new Random();
57
58     //Nothing to do until we're handed our first fork.
59   }
60
61   //A reference to the Dining machine that started it all off (to use as
       a 'printing proxy')
62   private Dining diningRoom;
63
64   public static final class ProvideFork implements Immutable {
65     public final Dining diningRoom;
66     public final int remainingPhilosophers; //The number of philosophers
          that must still be instantiated.
67     public final Fork nextFork;
68     public final Fork finalFork; //Notionally fork0. (the first
         philosopher's top priority fork, which will also be the final
         philosopher's highest priority fork)
69
70     public ProvideFork(Dining dining, int remainingPhilosophers, Fork
         next, Fork finalFork) {
71       this.remainingPhilosophers = remainingPhilosophers;
72       this.nextFork = next;
```

```
73        this.finalFork = finalFork;
74        this.diningRoom = dining;
75      }
76
77      @Override
78      public void flatten(DataOutputStream out) throws IOException {
79        out.writeInt(remainingPhilosophers);
80        Utilities.flatten(nextFork, out);
81        Utilities.flatten(finalFork, out);
82        Utilities.flatten(diningRoom, out);
83      }
84
85      public ProvideFork(DataInputStream in) throws NotInflatableException
            , IOException {
86        remainingPhilosophers = in.readInt();
87        nextFork = (Fork) Utilities.inflate(in);
88        finalFork = (Fork) Utilities.inflate(in);
89        diningRoom = (Dining) Utilities.inflate(in);
90      }
91    }
92
93    private final SecondForkResponseHandler secondForkHandler = new
          SecondForkResponseHandler();
94    private final class SecondForkResponseHandler implements Handler<
          Envelope<Boolean>> {
95      @Override
96      public void handle(Envelope<Boolean> info) {
97        if (info.getPayload()) {
98          //Got the second fork! We can eat.
99          eat();
100       } else {
101         //failed to gain the second fork.
102         // we can either release the first and try again from the
                beginning or just wait for this fork.
103
104         //try again for the second fork:
105         requestSecondFork();
106         //firstFork.releaseFork.send(Philosopher.this);
107         //think();
108       }
109     }
110   }
111
112   private final FirstForkResponseHandler firstForkHandler = new
          FirstForkResponseHandler();
113   private final class FirstForkResponseHandler implements Handler<
          Envelope<Boolean>> {
114     @Override
115     public void handle(Envelope<Boolean> info) {
```

```
116         if (info.getPayload()) {
117           //Got the first fork!
118           requestSecondFork();
119         } else {
120           //Try again:
121           requestFirstFork();
122         }
123       }
124     }
125 
126     /**
127      * Attempts to gain forks so that eating may occur.
128      */
129     private void requestFirstFork() {
130       firstFork.gainFork.query(Nothing.NOTHING, firstForkHandler);
131     }
132 
133     /**
134      * The length of time a philosopher eats for before releasing their
           forks.
135      */
136     public static final int EAT_DURATION_MS = 1000;
137     public static final int THINK_MAXDURATION_MS = 2000;
138 
139     private Random rng;
140 
141     /**
142      * We have the forks so eating may happen.
143      */
144     private void eat() {
145       //1) eat; wait a while before releasing forks. :)
146       final long eatingStarted = System.currentTimeMillis();
147 
148       String eatingMessage = System.currentTimeMillis()/1000+"s ph:"+ourID
             +": eating";
149 
150       diningRoom.eatingNotification.send(eatingMessage, new Handler<
             Nothing>() {
151         @Override
152         public void handle(Nothing n) {
153           long loggingDelay = System.currentTimeMillis()-eatingStarted;
154           long eatDuration = Math.min(EAT_DURATION_MS-loggingDelay, 0);
155 
156           //Eat for a while (so create a delay)
157           new Delay(eatDuration, new Handler<TimeEvent>() {
158             @Override
159             public void handle(TimeEvent info) {
160               //2) release forks
161               secondFork.releaseFork.send(Philosopher.this);
```

```
162            firstFork.releaseFork.send(Philosopher.this);
163            //3) Now think for some time.
164            think();
165          }
166        });
167      }
168    });
169  }
170
171  private void think() {
172    new Delay(rng.nextInt(THINK_MAXDURATION_MS), new Handler<TimeEvent
         >() {
173
174      @Override
175      public void handle(TimeEvent info) {
176        //4) thinking finished so we can just start again!
177        requestFirstFork();
178      }
179    });
180  }
181
182  private void requestSecondFork() {
183    secondFork.gainFork.query(Nothing.NOTHING, secondForkHandler);
184  }
185
186  /**
187   * this is how we get our forks
188   */
189  @Override
190  protected void setup(ProvideFork forks) {
191    ourID = forks.remainingPhilosophers;
192
193    diningRoom = forks.diningRoom;
194
195    if (forks.remainingPhilosophers==0) {
196      //We're the last philosopher...
197      firstFork = forks.finalFork;
198      secondFork = forks.nextFork;
199      //All setup!
200    } else {
201      //The fork the next philosopher will use as their high-priority
           fork.
202      Fork nextFork;
203      //The final philosopher's high-priority fork.
204      Fork finalFork;
205
206      if (forks.finalFork==null) {
207        //we're the first philosopher so we must create both forks...
208        firstFork = finalFork = newMachine(Fork.class);
```

```
209             secondFork = nextFork = newMachine(Fork.class);
210         } else {
211             //We're neither the first nor the last philosopher so we must
                    create one fork.
212
213             firstFork = forks.nextFork;
214             secondFork = nextFork = newMachine(Fork.class);
215             finalFork = forks.finalFork;
216         }
217
218         //We're not the last philosopher so there must be at least one
                more...
219         //provide the next (and newest philosopher) with a set of forks:
220         setupMachine(Philosopher.class, new ProvideFork(diningRoom, forks.
                remainingPhilosophers-1, nextFork, finalFork), null);
221
222         //we don't actually need a reference to our peer, so there's no
                handler for the machine reference.
223     }
224
225     //done with fork setup!
226     // time to EAT!
227     requestFirstFork();
228   }
229 }
```

Appendix D

Sample Experimental Log

The following log was captured during a series of experiments with the SpeedTestHP microbenchmark across many different processor configurations and using the sequential processor iterator. The output from Chi and the messages concerning the program download to the FPGA have been abridged as they are verbose and contribute little to the understanding of an experimental execution. The redundancy and regularity of the log experimental log format enables easy conversion of these captured logs into data tables and LaTeX plots.

```
##./executeapplication.sh# -network-size 5x3 -run -exec examples.speedtest.highperf.SpeedTestHP -target BLUESHELL  Sat Jul  4
      22:34:47 BST 2015
./timeout.sh -t 900 java -Xmx2G -jar chi.jar -save-temps -heapsize 49152 -force CLASSNAMES -network-size 5x3 -run -exec examples.
      speedtest.highperf.SpeedTestHP -target BLUESHELL -minimal -buildopts -Os 500 Sat Jul 4 22:34:47 BST 2015
Chi alpha-0.21.84 - Gary Plumbridge 2015
Features disabled: [STACKTRACES, ARITHMETICCHECKS, ARRAYBOUNDSCHECKS, ARRAYSTORECHECKS, ASSIGNMENTRULESCHECKS, CHECKCAST,
      NULLPOINTERCHECKS]
...
Application written.
Starting external build tool with command:
mb-gcc -mxl-multiply-high -mxl-barrel-shift -mno-xl-soft-mul -mno-xl-soft-div -fno-pic -mlittle-endian -DLITTLE_ENDIAN -mhard-float
       -mxl-float-convert -mxl-float-sqrt -mcpu=v8.50.b *.S -Wall -Wno-unused-but-set-variable -Wno-unused-label -fno-strict-
      aliasing -Wl,--script,lscript.ld -o a.out -Os *.c >/dev/null
Build succeeded.
Attempting to execute binary with command:
/home/gp/Dropbox/mjava/blueshell-experiments/bluetiles.sh /home/gp/Dropbox/mjava/blueshell-experiments/a.out
~/workspace/blueshell/tests/java ~/Dropbox/mjava/blueshell-experiments
-rw-r--r-- 1 gp gp 200K Jul  4 22:35 hw.0000c008.bin
   text    data     bss     dec     hex filename
 173846   30480     144  204470   31eb6 /home/gp/Dropbox/mjava/blueshell-experiments/a.out
~/Dropbox/mjava/blueshell-experiments
Using binary ../tests/java/hw.0000c008.bin
Loading FPGA bit file vc707.bit
Using board name vc707
Using net config ../netconfig.py
Virtual lab mode
Using auth file ../vc707.key
Running test procedure
Connecting to the board...
Sending bit file...
Programming FPGA...
```

```
Open UART...
Call <function myTestProc at 0x1b990c8>
Uploading binaries...
Read ../tests/java/hw.0000c008.bin...
../tests/java/hw.0000c008.bin uploaded to 0xc008
../tests/java/hw.0000c008.bin uploaded to 0xc108
...
../tests/java/hw.0000c008.bin uploaded to 0x3dd08
../tests/java/hw.0000c008.bin uploaded to 0x3de08
Saving processor coordinates into its ram...
Set memory[0xc000] = [USER_X, USER_Y] = [0, 0]
Read back: [0, 0]
Boot microblaze: 0000c008
0,0 alive!
0,1 ali0,2 alive!
1,2 alive!
2,2 alive!
3,2 alive!
ve!
1,1 a4,2 alive!
live!
2,1 alive!
3,1 alive!
4,1 alive!
1,0 alive!
2,0 alive!
3,0 alive!
4,0 alive!
SpeedTestHP@0,0:1> #i#SpeedTestHP#, processors, pairs, avgLatencyuS, rateB/s, messages
SpeedTestHP@0,0:1> #i#SpeedTestHP# Message size: 500
SpeedTestHP@0,0:1> Pairs: 0. No messages exchanged in last interval.
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:1) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 1, 7329, 54612, 1087
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:1) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 2, 7341, 122200, 2444
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:1) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 3, 7326, 190450, 3809
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:1) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 4, 7477, 253250, 5065
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:1) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 5, 7509, 318600, 6372
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:1) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 6, 7601, 380250, 7605
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:1) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 7, 7701, 439950, 8799
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:2) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 8, 8351, 465350, 9307
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:2) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 9, 9058, 484150, 9683
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:2) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 10, 9802, 498250, 9965
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:2) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 11, 10514, 512200, 10244
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:2) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 12, 11148, 527650, 10553
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:2) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 13, 11804, 540650, 10813
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:2) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 14, 12399, 554950, 11099
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:3) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 15, 13203, 559000, 11180
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:3) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 16, 13993, 563100, 11262
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:3) and PingServer created!
```

```
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 17, 14794, 566500, 11330
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:3) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 18, 15615, 569050, 11381
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:3) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 19, 16429, 570900, 11418
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:3) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 20, 17203, 574500, 11490
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:3) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 21, 17946, 578550, 11571
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:4) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 22, 18959, 574300, 11486
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:4) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 23, 20076, 566900, 11338
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:4) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 24, 21099, 562850, 11257
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:4) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 25, 22129, 559650, 11193
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:4) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 26, 23102, 557900, 11158
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:4) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 27, 24073, 555900, 11118
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:4) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 28, 25088, 553400, 11068
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:5) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 29, 26429, 544100, 10882
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:5) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 30, 27507, 541350, 10827
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:5) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 31, 28580, 538100, 10762
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:5) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 32, 29850, 532000, 10640
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:5) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 33, 30967, 529250, 10585
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:5) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 34, 32169, 524800, 10496
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:5) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 35, 33326, 521550, 10431
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:6) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 36, 34835, 513350, 10267
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:6) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 37, 35986, 511050, 10221
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:6) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 38, 37469, 503800, 10076
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:6) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 39, 38909, 498150, 9963
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:6) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 40, 40614, 489600, 9792
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:6) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 41, 41540, 490500, 9810
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:6) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 42, 43218, 483450, 9669
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:7) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 43, 44246, 483300, 9666
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:7) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 44, 45858, 477050, 9541
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:7) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 45, 47206, 474250, 9485
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:7) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 46, 48842, 468350, 9367
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:7) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 47, 50499, 463100, 9262
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:7) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 48, 52129, 458050, 9161
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:7) and PingServer created!
```

```
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 49, 53362, 457000, 9140
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:8) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 50, 55188, 451000, 9020
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:8) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 51, 56840, 446400, 8928
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:8) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 52, 59001, 438650, 8773
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:8) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 53, 60373, 437000, 8740
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,0:8) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 54, 62131, 432650, 8653
SpeedTestHP@0,0:1> New PingClient (PingClientHP@3,2:8) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 55, 64330, 425350, 8507
SpeedTestHP@0,0:1> New PingClient (PingClientHP@4,1:8) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 56, 66380, 420350, 8407
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,0:9) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 57, 68154, 416250, 8325
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,1:9) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 58, 70541, 409400, 8188
SpeedTestHP@0,0:1> New PingClient (PingClientHP@2,1:9) and PingServer created!
SpeedTestHP@0,0:1> #d#SpeedTest#, 15, 59, 72187, 406900, 8138
SpeedTestHP@0,0:1> New PingClient (PingClientHP@1,2:9) and PingServer created!
SpeedTestHP@0,0:1> Terminating due to timeout of (ms): 600000
Program terminated
Exit code: 80
Execution failed with error code: 1
Call graph dumped to callgraph.dot.
Type graph dumped to typegraph.dot.
Machine dependency graph dumped to examples.speedtest.highperf.SpeedTestHP.dot
Chi completed in 12m 22.978s
```

Appendix E

Scoped Memory Allocation in Chi

Region-Based Memory Management

Standard Java's model of fully automatic memory management is very easy to program with, but there are two important issues that make it undesirable in some embedded contexts:

1. Standard garbage collectors introduce unpredictable delays into program execution which can hinder or destroy the ability to reason about the timing of an application. This is especially important in embedded real-time systems where timing is a critical aspect of system functionality, and where processors may not have much spare computational capacity.

2. Garbage collection requires the ability to traverse the heap to discover reachable objects which implies additional runtime memory overheads. As with serialisation, metadata is required about which fields in a class are references so that they can be followed to other reachable objects.

3. Some mechanism to identify and deallocate unreachable objects is also required. This might involve recording all object allocations so that unreachable objects can be identified and deallocated, but the cost of recording several hundred object allocations (1KiB for a 256-length table of 32-bit pointers) is non-trivial in situations where there is only 8KiB of ram.

   Other schemes, such as semispace collectors[1] do not require a record of object

[1]Semispace GC schemes copy reachable objects to a fresh heap and then discards the original heap along with all of the unreachable objects it contains

allocations but do require a reserved space in which to copy live objects during a collection. Where the heap is divided into two equal parts a semispace GC implies a spatial overhead of at least 50%.

Real-time garbage collectors have been investigated thoroughly [16, 20, 108, 159, 160, 142, 7, 54, 188] to address the unpredictable timing characteristics, but the requisite memory overheads are still substantial for processors with very small quantities of ram.

Method	Description
Scope createScope(**int**)	Creates a new memory scope with the size specified and returns the descriptor Scope object. The new scope's parent is the active scope. Does not change the current allocation context.
void destroyScope(Scope)	Deallocates the specified scope. All objects within the scope are lost. If there are other scopes in existence that are children of the specified scope then this will throw a ScopeHasChildrenError.
void executeInScope(Scope, Handler<T>, T)	Executes the handler specified with the provided parameter in the allocation context provided. Exceptions thrown by the handler will be re-thrown as a ThrowBoundaryError. If the specified scope is null then the handler will be executed in the immortal allocation context.
void executeInScope(Scope, Handler<Void>)	As above but for handlers that do not expect a parameter.
void transientScope(**int**, Handler<T>, T)	Creates a new transient scope with the specified size for the duration of the execution of the handler supplied. The handler will be executed in the transient allocation context with the parameter supplied and when the handler returns the scope will be destroyed.
void transientScope(**int**, Handler<Void>)	As above but for handlers that do not expect a parameter.
Scope getActiveScope()	Returns the Scope descriptor for the current allocation context. This should not be used in transient scopes.

Table E.1: *Network-Chi's scope-based memory management API. These are all static methods of the ScopedMemoryManager class.*

Where it's inappropriate to use garbage collection, Chi provides a region based memory management API which avoids the need for metadata, records of objects and also has extremely predictable timing. A summary of the Network-Chi memory management API can be seen in table E.1. This API is essentially a much simplified version of the Real-Time Specification for Java (RTSJ)[219] model but with a number of compromises made to further reduce the runtime overhead of the framework.

In Network-Chi objects are allocated into either the *immortal* memory region or into a memory *scope*. Objects are allocated in the usual way with the `new` keyword and the region that they are allocated into depends on the currently active *allocation context*. Memory used by objects allocated into the immortal region can never be reclaimed for the entire lifetime of the application, but memory used by objects allocated into a scope can be reclaimed by destroying the scope. Scopes have a fixed size that is exclusively allocated to them when they are created, and this memory is reclaimed in full when they are destroyed, regardless of the number and size of the objects contained within.

Some important differences between the RTSJ memory model and Network-Chi include:

- When the scoped memory manager is in use, there is no normal garbage collected heap. At system startup the allocation context is the immortal region.
- In RTSJ the `ScopedMemory` objects used to refer to a memory scope are distinct from the actual *backing store* of memory that they refer to. RTSJ allows for a backing store to become inactive and reclaimed independently of the `ScopedMemory` object that refers to it, likewise freeing the backing store does not free the `ScopedMemory` object as that is allocated in the context that constructed it.

  Network-Chi uses a simpler model whereby the `Scope` object is allocated within the backing store it refers to and has exactly the same lifetime as the backing store.
- Network-Chi memory scopes that were explicitly created by the programmer are also manually deallocated by the programmer. This means that unlike the RTSJ, Network-Chi's memory model is **unsafe**. It is possible if the programmer is careless to retain a reference to an object that is no longer valid in memory (a dangling reference), potentially leading to unpredictable and almost certainly incorrect behaviour. RTSJ-style assignment rules checks are performed by the runtime but cannot avoid dangling references in two situations:

1. A reference to an object in a scope is stored in a local variable (which are not subject to assignment rules checks) and then the scope is destroyed. The reference is now dangling.
2. A reference to a `Scope` object is retained after the scope it refers to is destroyed. `Scope` descriptor objects in Network-Chi are never subject to assignment rules enforcement.

- Network-Chi only provides linear time allocation of scoped memory. The backing store is allocated and zeroed immediately on scope creation.

To partially mitigate the risk of dangling references the memory API contains methods for the use of anonymous, transient scopes. This caters for the common use case where an operation is to be performed with no net consumption of memory, but where it cannot be guaranteed that all library functions do not allocate temporary objects. When transient scopes are used they are entirely safe as assignment rules enforcement prevents references escaping to less deeply nested scopes and immortal memory, and transient scopes cannot return data.

The `ScopedMemoryManager` is implemented in Java and in addition to the programmer visible scope management API it also supplies two important runtime methods used exclusively by the compiler. The `allocate` method is called by the compiler every time an object is constructed to reserve space for the new object in the current allocation context, and the `checkAssignment` method is called by the compiler every time a reference is assigned into an object or array. It is obviously extremely expensive to check the validity of every reference assignment but this is required to prevent references to more deeply nested scopes from being retained beyond the lifetime of the scope. Using static analysis to reduce the runtime burden of assignment rules checks is an area of ongoing research, especially for Safety Critical Java[203] which also uses a scoped memory model.

Appendix F

Glossary

AOT Ahead-of-Time

API Application Programming Framework

ATC Asynchronous Transfer of Control

CAN Controller Area Network

CBO Concrete Binary Object

Chi Concrete Hardware Implementation

FPGA Field-Programmable Gate Array

FPU Floating Point Unit

GALS Globally Asynchronous Locally-Synchronous

HDL Hardware Description Language

HLL High Level Language

HPF Highest Priority First

IC Integrated Circuit

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

J2ME Java 2 Micro Edition

JCP Java Community Process

JIT Just-in-Time

JRE Java Runtime Environment

JSR Java Specification Requests

JVM Java Virtual Machine

KPN Kahn Process Networks

LIN Local Interconnect Network

MAA Machine Abstract Architecture

MOSFET Metal Oxide Semiconductor Field Effect Transistor

MPNoC Multi-processor Network on Chip

NFV Network Function Virtualisation

PCG Processor Connectivity Graph

RTC Real-Time Clock

RTL Register Transfer Level

SCC Single Chip Cloud

SDF Synchronous Data Flow

SEM Simple Ephemeral Machine

SPI Serial Peripheral Interface

TCP Transmission Control Protocol

TDMA Time Division Multiple Access

TPIF Two Party Interaction Framework

UDP User Datagram Protocol

WCET Worst Case Execution Time

WCMC Worst Case Memory Consumption

WSN Wireless Sensor Network

Bibliography

[1] Online Encyclopedia Of Integer Sequences: A006877. URL http://oeis.org/A006877.

[2] GCC Reference Manual: Using the GNU Compiler Collection (GCC). URL https://gcc.gnu.org/onlinedocs/gcc/.

[3] Ptolemy II Project, 2012. URL http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm.

[4] D. Abts, S. Scott, and D.J. Lilja. So many states, so little time: verifying memory coherence in the Cray X1. In *Proceedings International Parallel and Distributed Processing Symposium*, page 10. IEEE Comput. Soc, 2003. ISBN 0-7695-1926-1. doi: 10.1109/IPDPS.2003.1213087.

[5] Gul A Agha. Actors: A model of concurrent computation in distributed systems. (AI-TR-844). Technical report, Massachusetts Inst Of Tech Cambridge Artificial Intelligence Lab, DTIC Document, 1985.

[6] Agility. *Handel-C Language Reference Manual*. Agility Design Solutions Inc., 2008.

[7] Aicas GmbH. JamaicaVM — Java Technology for Realtime, 2013. URL http://www.aicas.com/jamaica-pe.html.

[8] Altera Corporation. Documentation: Nios II Processor, 2013. URL http://www.altera.com/literature/lit-nio2.jsp.

[9] Jeppe L. Andersen, Mikkel Todberg, Andreas E. Dalsgaard, and René Rydhof Hansen. Worst-case memory consumption analysis for SCJ. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems — JTRES '13*, pages 2–10. ACM Press, oct 2013. ISBN 9781450321662. doi: 10.1145/2512989.2513000.

[10] Apache Software Foundation. Apache Commons BCEL. URL http://commons.apache.org/bcel/.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A Yelick. The Landscape of Parallel Computing Research: A View from Berkeley, dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[12] Neil Ashby. Relativity in the Global Positioning System. *Living Reviews in Relativity*, 6, 2003. ISSN 1433-8351. doi: 10.12942/lrr-2003-1. URL http://relativity.livingreviews.org/Articles/lrr-2003-1/.

[13] Peter J. Ashenden. *The Designer's Guide to VHDL, Second Edition (Systems on Silicon)*. Morgan Kaufmann, 2001. ISBN 1558606742.

[14] Atego. Aonix Perc Pico. URL http://www.atego.com/products/aonix-perc-pico/.

[15] Atmel. ATmega128 8-Bit AVR RISC Microcontroller, 2012. URL http://www.atmel.com.

[16] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend. In *Proceedings of the 7th ACM international conference on Embedded software - EMSOFT '08*, page 245, New York, New York, USA, oct 2008. ACM Press. ISBN 9781605584683. doi: 10.1145/1450058.1450092.

[17] C Austin and M Pawlan. JNI Technology. In *Advanced Programming for the Java 2 Platform*, pages 207–230. 2000. URL http://java.sun.com/docs/books/jni/download/jni.pdf.

[18] AUTOSAR Consortium. AUTOSAR - 4.2.1 - TPS - System Template. Technical report, 2015. URL http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/templates/standard/AUTOSAR{_}TPS{_}SystemTemplate.pdf.

[19] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. ISSN 00045411. doi: 10.1145/4221.4227.

[20] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '03*, volume 38, pages 285–298, New York, New York, USA, jan 2003. ACM Press. ISBN 1581136285. doi: 10.1145/604131.604155.

[21] DF Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, 1997.

[22] R. Baert, E. Brockmeyer, S. Wuytack, and T.J. Ashby. Exploring parallelizations of applications for MPSoC platforms using MPA. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 1148–1153. IEEE, apr 2009. ISBN 978-1-4244-3781-8. doi: 10.1109/DATE.2009.5090836.

[23] Henry Baker and Carl Hewitt. Laws for Communicating Parallel Processes, may 1977. URL `http://dspace.mit.edu/handle/1721.1/41962`.

[24] T.P. Baker and G.A. Riccardi. Ada Tasking: From semantics to Efficient Implementation. *IEEE Software*, 2(2):34–46, mar 1985. ISSN 0740-7459. doi: 10.1109/MS.1985.230349.

[25] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to Better - How to Make Bitcoin a Better Currency. In AngelosD. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 399–414. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32945-6. doi: 10.1007/978-3-642-32946-3-29.

[26] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002. ISSN 00189162. doi: 10.1109/2.976921.

[27] Tim Berners-Lee, Roy Fielding, and Larry Masinter. RFC 3986: Uniform resource identifier (uri): Generic syntax. *Internet Engineering Task Force — Request for Comments*, 2005.

[28] Bluespec Inc. Bluespec System Verilog (BSV) URL:http://www.bluespec.com/products, 2013.

[29] Huang Bo, Dong Hui, Wang Dafang, and Zhao Guifan. Basic Concepts on AUTOSAR Development. In *2010 International Conference on Intelligent Computation*

Technology and Automation, volume 1, pages 871–873. IEEE, may 2010. ISBN 978-1-4244-7279-6. doi: 10.1109/ICICTA.2010.571.

[30] Hans-J. Boehm. Destructors, finalizers, and synchronization. *ACM SIGPLAN Notices*, 38(1):262–272, jan 2003. ISSN 03621340. doi: 10.1145/640128.604153.

[31] M Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE*, 12:11–13, 2007. ISSN 1098-4232. doi: 10.1109/N-SSC.2007.4785534.

[32] M.T. Bohr. Interconnect scaling — The real limiter to high performance ULSI. In *Proceedings of International Electron Devices Meeting*, pages 241–244. IEEE. ISBN 0-7803-2700-4. doi: 10.1109/IEDM.1995.499187.

[33] Christian Bradatsch, Florian Kluge, and Theo Ungerer. A cross-domain system architecture for embedded hard real-time many-core systems. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 2034–2041. IEEE, nov 2013. ISBN 978-0-7695-5088-6. doi: 10.1109/HPCC.and.EUC.2013.293.

[34] Robert Braden. RFC-1122: Requirements for internet hosts. *Request for Comments*, pages 356–363, 1989.

[35] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16, 1998.

[36] Dai Bui, Edward Lee, Isaac Liu, Hiren Patel, and Jan Reineke. Temporal isolation on multiprocessing architectures. In *Proceedings of the 48th Design Automation Conference — DAC '11*, page 274, New York, New York, USA, jun 2011. ACM Press. ISBN 9781450306362. doi: 10.1145/2024724.2024787.

[37] Alan Burns and Andy Wellings. *Real-time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX, 4th Ed.* Addison Wesley, 2008. ISBN 0201729881.

[38] Rod Burstall. Christopher Strachey – Understanding Programming Languages. *Higher-Order and Symbolic Computation*, 13(1-2):51–55. ISSN 1573-0557. doi: 10.1023/A:1010052305354.

[39] Everton A. Carara, Roberto P. de Oliveira, Ney L. V. Calazans, and Fernando G. Moraes. HeMPS — A framework for NoC-based MPSoC generation. In *2009 IEEE International Symposium on Circuits and Systems*, pages 1345–1348. IEEE, may 2009. ISBN 978-1-4244-3827-3. doi: 10.1109/ISCAS.2009.5118013.

[40] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985. ISSN 03600300. doi: 10.1145/6041.6042.

[41] J.M.P. Cardoso and H.C. Neto. Towards an automatic path from Java bytecodes to hardware through high-level synthesis. In *IEEE International Conference on Electronics, Circuits and Systems*, volume 1, pages 85–88. Citeseer, 1998. doi: 10.1.1.37.398.

[42] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An integrated framework for MPSoC application parallelization. In *Design Automation Conference*, pages 754–759, New York, New York, USA, jun 2008. ACM Press. ISBN 9781605581156. doi: 10.1109/DAC.2008.4555920.

[43] Daniel M Chapiro. Globally-Asynchronous Locally-Synchronous Systems. (Doctoral Thesis). Technical report, DTIC Document, 1984.

[44] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementationâĂŤA performance view. *IBM Journal of Research and Development*, 51(5):559–572, sep 2007. ISSN 0018-8646. doi: 10.1147/rd.515.0559.

[45] E. Cheong. galsC: A Language for Event-Driven Embedded Systems. In *Design, Automation and Test in Europe*, pages 1050–1055. IEEE, 2005. ISBN 0-7695-2288-2. doi: 10.1109/DATE.2005.165.

[46] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. TinyGALS. In *Proceedings of the 2003 ACM symposium on applied computing — SAC '03*, page 698, New York,

New York, USA, mar 2003. ACM Press. ISBN 1581136242. doi: 10.1145/952532.952668.

[47] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, C Cui, H Deng, and Others. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, pages 22–24, 2012.

[48] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 155–166. IEEE, 2011.

[49] Michael D. Ciletti. *Modeling, Synthesis, and Rapid Prototyping with the VERILOG HDL*. Prentice Hall, 1999. ISBN 0139773983. URL `http://www.amazon.com/Modeling-Synthesis-Rapid-Prototyping-VERILOG/dp/0139773983`.

[50] David E Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *Embedded Software*, pages 114–130. Springer, 2001.

[51] S. de Deugd, R. Carroll, K. Kelly, B. Millett, and J. Ricker. SODA: Service Oriented Device Architecture. *IEEE Pervasive Computing*, 5(3):94–96, c3, jul 2006. ISSN 1536-1268. doi: 10.1109/MPRV.2006.59.

[52] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51 (1):107, jan 2008. ISSN 00010782. doi: 10.1145/1327452.1327492.

[53] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideovt, Ernest Bassous, and Andre R. Leblanc. Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions. *IEEE Solid-State Circuits Newsletter*, 12(1), 2007. ISSN 1098-4232. doi: 10.1109/N-SSC.2007.4785543.

[54] D Detlefs. A hard look at hard real-time garbage collection. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 23–32, 2004. doi: 10.1109/ISORC.2004.1300325.

[55] Allen Dewey and Anthony Gadient. VHDL Motivation. *IEEE Design & Test of Computers*, 3(2):12–16, 1986. ISSN 0740-7475. doi: 10.1109/MDT.1986.294898.

[56] Ray D'Inverno. *Introducing Einstein's Relativity*. Clarendon Press, 1992. ISBN 0198596863.

[57] Sophia Drossopoulou, Dave Clarke, and James Noble. Types for Hierarchic Shapes. In *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 1–6. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-33095-0. doi: 10.1007/11693024-1.

[58] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In ErnestF. Brickell, editor, *Advances in Cryptology — CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer Berlin Heidelberg, 1993. ISBN 978-3-540-57340-1. doi: 10.1007/3-540-48071-4-10.

[59] Eclipse Foundation. JDT — Java development tools, 2015. URL `https://projects.eclipse.org/projects/eclipse.jdt`.

[60] D. Edenfeld, A.B. Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors. *Computer*, 37(1):47–56, jan 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.1260725.

[61] Albert Einstein. *Relativity: The special and the general theory*. Princeton University Press, 1916.

[62] J. Eker, J.W. Janneck, E.A. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. Taming heterogeneity — The Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, jan 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805829. URL `http://ieeexplore.ieee.org/xpl/freeabs{_}all.jsp?arnumber=1173203`.

[63] Michael D Ernst. Type Annotations (JSR 308) and the Checker Framework. URL `http://types.cs.washington.edu/jsr308/`.

[64] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, and Others. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[65] Kang Su Gatlin. Trials and tribulations of debugging concurrency. *ACM Queue*, 2 (7):66, oct 2004. ISSN 15427730. doi: 10.1145/1035594.1035623.

[66] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Technical report, 1996. URL https://hal.inria.fr/inria-00073732/.

[67] GNU. GCC: The GNU Compiler Collection, 2015. URL http://gcc.gnu.org/.

[68] GNU. The GNU Compiler for the Java Programming Language, 2015. URL http://gcc.gnu.org/java/.

[69] Diana Gohringer, Michael Hubner, Volker Schatz, and Jurgen Becker. Runtime adaptive multi-processor system-on-chip: RAMPSoC. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7. IEEE, apr 2008. ISBN 978-1-4244-1693-6. doi: 10.1109/IPDPS.2008.4536503.

[70] Diana Gohringer, Michael Hubner, Laure Hugot-Derville, and Jurgen Becker. Message Passing Interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 357–364. IEEE, jul 2010. ISBN 978-1-4244-7936-8. doi: 10.1109/ICSAMOS.2010.5642043.

[71] G Goos and J Hartmanis. *The Programming Language Ada: Reference Manual. Proposed Standard Document United States Department of Defense*. Lecture Notes in Computer Science. Springer, 1 edition, 1983. ISBN 3540106936. URL http://www.amazon.com/Programming-Language-Ada-Reference-Department/dp/3540106936.

[72] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Ruadulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. Guaranteeing the quality of services in networks on chip. In *Networks on chip*, pages 61–82. Springer, 2003.

[73] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201703238.

[74] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java*

Language Specification, Java SE 7 Edition. 2011. URL http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf.

[75] Ian Gray and Neil C. Audsley. Supporting islands of coherency for highly-parallel embedded architectures using compile-time virtualisation. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems - SCOPES '10*, page 1, New York, New York, USA, jun 2010. ACM Press. ISBN 9781450300841. doi: 10.1145/1811212.1811223.

[76] Ian Gray, Gary Plumbridge, and Neil C. Audsley. Toolchain-based approach to handling variability in embedded multiprocessor system on chips. *IET Computers & Digital Techniques*, 9(1):82–92, jan 2015. ISSN 1751-8601. doi: 10.1049/iet-cdt.2014.0070.

[77] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, feb 2009. ISSN 03043975. doi: 10.1016/j.tcs.2008.09.019.

[78] D.S. Hardin. *Real-time objects on the bare metal: an efficient hardware realization of the Java Virtual Machine*. IEEE Comput. Soc, 2001. ISBN 0-7695-1089-2. doi: 10.1109/ISORC.2001.922817.

[79] T. Harmon and R. Klefstad. A Survey of Worst-Case Execution Time Analysis for Real-Time Java. *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007. doi: 10.1109/IPDPS.2007.370422.

[80] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., sep 2011. ISBN 012383872X, 9780123838728.

[81] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, aug 1973.

[82] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, oct 1974. ISSN 00010782. doi: 10.1145/355620.361161. URL http://portal.acm.org/citation.cfm?doid=355620.361161.

[83] CAR Hoare. Communicating sequential processes. *Communications of the ACM*, 21 (8):677, 1978.

[84] Jim Holt, Anant Agarwal, Sven Brehmer, Max Domeika, Patrick Griffin, and Frank Schirrmeister. Software Standards for the Multicore Era. *IEEE Micro*, 29(3):40–51, may 2009. ISSN 0272-1732. doi: 10.1109/MM.2009.48.

[85] M. Horowitz and W. Dally. How scaling will change processor architecture. In *2004 IEEE International Solid-State Circuits Conference (IEEE Cat. No.04CH37519)*, pages 132–133. IEEE, 2004. ISBN 0-7803-8267-6. doi: 10.1109/ISSCC.2004.1332629.

[86] Jason Howard, Saurabh Dighe, Sriram R. Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, Guido Droege, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek K. De, and Rob Van Der Wijngaart. A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, 2011. ISSN 00189200. doi: 10.1109/JSSC.2010.2079450.

[87] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *Proceedings of the 2003 conference on Asia South Pacific design automation - ASPDAC*, page 233. ACM Press, jan 2003. ISBN 0780376609. doi: 10.1145/1119772.1119818.

[88] Libo Huang, Zhiying Wang, and Nong Xiao. Accelerating NoC-Based MPI Primitives via Communication Architecture Customization. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 141–148. IEEE, jul 2012. ISBN 978-1-4673-2243-0. doi: 10.1109/ASAP.2012.33.

[89] IBM. developerWorks : IBM developer kits : Downloads, jun 2011. URL `http://www.ibm.com/developerworks/java/jdk/`.

[90] IEEE. IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7. pages c1–3826, dec 2008. doi: 10.1109/IEEESTD.2008.4694976.

[91] Typesafe inc. Akka (Java and Scala concurrency framework), 2015. URL `http://akka.io`.

[92] Leandro Soares Indrusiak and Manfred Glesner. An Actor-Oriented Model-Based Design Flow for Systems-on-Chip. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme II*, page 65.

[93] Leandro Soares Indrusiak, Luciano Ost, Leandro Möller, Fernando Moraes, and Manfred Glesner. Applying UML Interactions and Actor-Oriented Simulation to the Design Space Exploration of Network-on-Chip Interconnects. In *2008 IEEE Computer Society Annual Symposium on VLSI*, pages 491–494. IEEE, 2008. ISBN 978-0-7695-3291-2. doi: 10.1109/ISVLSI.2008.20.

[94] Infineon Technologies AG. AURIX Family — TC27xT — Infineon Technologies. URL http://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/aurix-tm-family-{%}E2{%}80{%}93-tc27xt/channel.html?channel=db3a30433cfb5caa013d01df64d92edc.

[95] INMOS Ltd. *Occam 2 reference manual*, volume 12. Prentice-Hall, jun 1989. ISBN 0-13-629321-3. doi: 10.1016/0167-6423(89)90030-0. URL http://linkinghub.elsevier.com/retrieve/pii/0167642389900300.

[96] Intel. Intel Xeon Phi Coprocessor Applications and Solutions Catalog. Technical Report 5 - January, 2016. URL https://software.intel.com/en-us/xeonphicatalog.

[97] ISO. ISO/IEC 9899 — Programming Languages — C, 1999.

[98] ISO. Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling. ISO 11898-1:2003, International Organization for Standardization, Geneva, Switzerland, 2003.

[99] ISO. Tractors and machinery for agriculture and forestry — Serial control and communications data network — Part 1: General standard for mobile data communication. ISO 11783-1:2007, International Organization for Standardization, Geneva, Switzerland, 2007. URL http://www.iso.org/iso/catalogue{_}detail.htm?csnumber=39122.

[100] ISO. Road vehicles — FlexRay communications system — Part 1: General infor-

mation and use case definition. ISO 17458-1:2013, International Organization for Standardization, Geneva, Switzerland, 2013.

[101] S.A. Ito, L. Carro, and R.P. Jacobi. Making Java work for microcontroller applications. *IEEE Design & Test of Computers*, 18(5):100–110, 2001. ISSN 07407475. doi: 10.1109/54.953277.

[102] A. Jantsch. Models of Computation for Networks on Chip. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 165–178. IEEE. ISBN 0-7695-2556-3. doi: 10.1109/ACSD.2006.14.

[103] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[104] Mike Williams Joe Armstrong, Robert Virding, Claes Wikstrom. *Concurrent Programming in ERLANG*. Prentice Hall, 2nd edition, 1996. ISBN 0-13-508301-X.

[105] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[106] Jaume Joven, Oriol Font-Bach, David Castells-Rufas, Ricardo Martinez, Lluis Teres, and Jordi Carrabina. xENoC — An eXperimental network-on-chip environment for parallel distributed computing on NoC-based MPSoC architectures. *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 141–148, feb 2008. doi: 10.1109/PDP.2008.24.

[107] G Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475, Stockholm, Sweden, 1974. North Holland, Amsterdam.

[108] Tomas Kalibera. Replicating real-time garbage collector for Java. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems - JTRES '09*, page 100, New York, New York, USA, sep 2009. ACM Press. ISBN 9781605587325. doi: 10.1145/1620405.1620420.

[109] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core

accelerator. *ACM SIGARCH Computer Architecture News*, 37(3):140, jun 2009. ISSN 01635964. doi: 10.1145/1555815.1555774.

[110] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978. ISBN 0131101633.

[111] D E Knuth. The Art of Computer Programming, volume 1: Fundamental Algorithms. 1968.

[112] H Kopetz. On the fault hypothesis for a safety-critical real-time system. In *Automotive Software — Connected Services in Mobile Networks*, volume 4147 of *Lecture Notes in Computer Science*, pages 31–42. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-37677-4. doi: 10.1007/11823063-3.

[113] Jeffrey C. Lagarias. The 3x+1 problem: An annotated bibliography (1963–1999). page 65, sep 2003. URL `http://arxiv.org/abs/math.NT/0309224`.

[114] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978. ISSN 00010782. doi: 10.1145/359545.359563.

[115] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, sep 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439.

[116] E.A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, may 2006. ISSN 0018-9162.

[117] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1458143`.

[118] EA Lee, Stephen Neuendorffer, and MJ Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12 (3):231–260, 2003.

[119] Man-Kit Leung. An extensible and retargetable code generation framework for actor models. Technical report, Technical report, EECS Department, University of California at Berkeley, 2009.

[120] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *2009 30th IEEE Real-Time Systems Symposium*, pages 57–67. IEEE, dec 2009. ISBN 978-0-7695-3875-4. doi: 10.1109/RTSS.2009.32.

[121] LIN Consortium. LIN Specification Package Revision 2.2A. Technical report, 2010.

[122] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Prentice Hall, second edition, apr 1999. ISBN 0201432943.

[123] S.M. Loo, B.E. Wells, N. Freije, and J. Kulick. *Handel-C for rapid prototyping of VLSI coprocessors for real time systems*. IEEE, 2002. ISBN 0-7803-7339-1. doi: 10.1109/SSST.2002.1026994. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1026994.

[124] Zhonghai Lu and Raimo Haukilahti. NoC Application Programming Interfaces. In *Networks on C*, chapter 12, pages 239–260. Kluwer Academic Publishers, jan 2003. ISBN 978-1-4020-7392-2. doi: 10.1007/0-306-48727-6-12.

[125] Philipp Mahr, Christian Lörchner, Harold Ishebabi, and Christophe Bobda. SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips. In *2008 International Conference on Reconfigurable Computing and FPGAs*, pages 187–192. IEEE, dec 2008. ISBN 978-1-4244-3748-1. doi: 10.1109/ReConFig.2008.27.

[126] Radu Marculescu. Contention-aware application mapping for Network-on-Chip communication architectures. In *2008 IEEE International Conference on Computer Design*, pages 164–169. IEEE, oct 2008. ISBN 978-1-4244-2657-7. doi: 10.1109/ICCD.2008.4751856.

[127] G. Martin. Overview of the MPSoC design challenge. In *43rd ACM/IEEE Design Automation Conference*, pages 274–279. IEEE, 2006. ISBN 1-59593-381-6. doi: 10.1109/DAC.2006.229245.

[128] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill Education, 4 edition, 2010. ISBN 0073191469.

[129] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78, jul 2012. ISSN 00010782. doi: 10.1145/2209249.2209269.

[130] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[131] T G Mattson, R F Van der Wijngaart, M Riepen, T Lehnig, P Brett, W Haas, P Kennedy, J Howard, S Vangal, N Borkar, G Ruhl, and S Dighe. The 48-core SCC processor: the programmer's view. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2010*, 2010.

[132] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip Terascale Processor. In *2008 SC — International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, nov 2008. ISBN 978-1-4244-2834-2. doi: 10.1109/SC.2008.5213921.

[133] David May. *The XMOS XS1 architecture*. 2009. URL `https://download.xmos.com/XM-000280-RF-2.pdf`.

[134] Sun Microsystems. Java ME Technology APIs & Docs, 2010. URL `http://java.sun.com/javame/reference/apis.jsp`.

[135] Jean-Yves Mignolet and Roel Wuyts. Embedded multiprocessor systems-on-chip programming. *IEEE software*, (3):34–41, 2009.

[136] Wajid Hassan Minhass, Johnny Öberg, and Ingo Sander. Design and implementation of a plesiochronous multi-core 4x4 network-on-chip FPGA platform with MPI HAL support. In *Proceedings of the 6th FPGAworld Conference on — FPGAworld '09*, pages 52–57, New York, New York, USA, sep 2009. ACM Press. ISBN 9781605588797. doi: 10.1145/1667520.1667527.

[137] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93, oct 2004. ISSN 01679260. doi: 10.1016/j.vlsi.2004.03.003.

[138] Guido Moritz, Claas Cornelius, Frank Golatowski, Dirk Timmermann, and Regina Stoll. Differences and Commonalities of Service-Oriented Device Architectures, Wireless Sensor Networks and Networks-on-Chip. In *2009 International Conference on Advanced Information Networking and Applications Workshops*, pages 482–487. IEEE, may 2009. ISBN 978-1-4244-3999-7. doi: 10.1109/WAINA.2009.31.

[139] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000) (Cat. No. PR00586)*, pages 52–59. IEEE Comput. Soc, 2000. ISBN 0-7695-0586-4. doi: 10.1109/ASYNC.2000.836791.

[140] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. URL `https://ww.bitcoin.co.jp/bitcoin.pdf`.

[141] Julio C. Navas and Tomasz Imielinski. GeoCast—geographic addressing and routing. In *Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking - MobiCom '97*, pages 66–76, New York, New York, USA, sep 1997. ACM Press. ISBN 0897919882. doi: 10.1145/262116.262132.

[142] A. Nilsson. *Compiling Java for real-time systems*. Licentiate thesis, Lund University, 2004.

[143] Leonhard Nobach and David Hausheer. Open, elastic provisioning of hardware acceleration in NFV environments. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–5. IEEE, mar 2015. ISBN 978-1-4799-5804-7. doi: 10.1109/NetSys.2015.7089057.

[144] NXP Semiconductors. NXP LPC1700 series MCUs. URL `http://www.nxp.com/products/microcontrollers/product{_}series/lpc1700/`.

[145] Jabulani Nyathi, Souradip Sarkar, and Partha Pratim Pande. Multiple clock domain synchronization for network on chip architectures. In *2007 IEEE International SOC Conference*, pages 291–294. IEEE, 2007. ISBN 978-1-4244-1592-2. doi: 10.1109/SOCC.2007.4545477.

[146] J.M. O'Connor and Marc Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997. URL `http://www.enee.umd.edu/class/enee698b.S2000/papers/oconnor97.pdf`.

[147] Kunle Olukotun, Basem A Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *ACM Sigplan Notices*, 31(9):2–11, 1996.

[148] Oracle Inc. Java Platform, Standard Edition 7 API Specification, . URL http://docs.oracle.com/javase/7/docs/api/overview-summary.html.

[149] Oracle Inc. java.util.concurrent (Java Platform SE 7), . URL http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html.

[150] Oracle Inc. NetBeans IDE, . URL https://netbeans.org/.

[151] Oracle Inc. OpenJDK, . URL http://openjdk.java.net/.

[152] Oracle Inc. Java Software, . URL https://www.oracle.com/java/index.html.

[153] Stephen Ostermiller. OstermillerUtils Java Utilities — MD5, 2012. URL http://ostermiller.org/utils/MD5.html.

[154] Matthew Felice Pace. BSP vs MapReduce. *Procedia Computer Science*, 9:246–255, 2012. ISSN 18770509. doi: 10.1016/j.procs.2012.04.026.

[155] José C S Palma, Leandro Soares Indrusiak, Fernando G Moraes, Alberto Garcia Ortiz, Manfred Glesner, and Ricardo A L Reis. *Adaptive Coding in Networks-on-Chip: Transition Activity Reduction Versus Power Overhead of the Codec Circuitry. (Chapter of Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation)*, pages 603–613. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-39097-8. doi: 10.1007/11847083-59.

[156] Preeti Ranjan Panda. SystemC-a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pages 75–80. IEEE, 2001.

[157] Partha Pratim Pande, Cristian Grecu, Michael Jones, André Ivanov, and Resve Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, 2005. ISSN 00189340. doi: 10.1109/TC.2005.134.

[158] Anastasios Papagiannis and Dimitrios S. Nikolopoulos. Hybrid address spaces: A methodology for implementing scalable high-level programming models on non-coherent many-core architectures. *Journal of Systems and Software*, 97:47–64, nov 2014. ISSN 01641212. doi: 10.1016/j.jss.2014.06.058.

[159] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, 43(6):33, may 2008. ISSN 03621340. doi: 10.1145/1379022.1375587.

[160] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism. *ACM SIGPLAN Notices*, 45(6):146, may 2010. ISSN 03621340. doi: 10.1145/1809028.1806615.

[161] David Plainfossé and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. In *IWMM '95 Proceedings of the International Workshop on Memory Management*, pages 211–249. Springer-Verlag, sep 1995. ISBN 3-540-60368-9.

[162] Gary Plumbridge and Neil Audsley. Extending Java for Heterogeneous Embedded System Description. In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–6. IEEE, jun 2011. ISBN 978-1-4577-0640-0. doi: 10.1109/ReCoSoC.2011.5981527.

[163] Gary Plumbridge and Neil Audsley. Translating Java for Resource Constrained Embedded Systems. *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, jul 2012. doi: 10.1109/ReCoSoC.2012.6322868.

[164] Gary Plumbridge and Neil C. Audsley. Programming FPGA based NoCs with Java. *2013 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2013*, 2013. doi: 10.1109/ReConFig.2013.6732323.

[165] Gary Plumbridge, Jack Whitham, and Neil Audsley. Blueshell: a platform for rapid prototyping of multiprocessor NoCs and accelerators. *ACM SIGARCH Computer Architecture News*, 41(5):107–117, 2014.

[166] Katalin Popovici, Frédéric Rousseau, Ahmed A. Jerraya, and Marilyn Wolf. *Embedded Software Design and Programming of Multiprocessor System-on-Chip*. Springer New York, New York, NY, 2010. ISBN 978-1-4419-5566-1. doi: 10.1007/978-1-4419-5567-8.

[167] Jon Postel and Others. RFC 768: User Datagram Protocol. *Internet Engineering Task Force — Request for Comments*, 1980.

[168] Jon Postel and Others. RFC 792: Internet control message protocol. *Internet Engineering Task Force — Request for Comments*, 1981.

[169] Jon Postel and Others. RFC 793: Transmission Control Protocol. *Internet Engineering Task Force — Request for Comments*, 1981.

[170] Jon Postel and Others. RFC 791: Internet protocol. *Internet Engineering Task Force — Request for Comments*, 1981.

[171] Jacob Postman, Tushar Krishna, Christopher Edmonds, Li-Shiuan Peh, and Patrick Chiang. Swift: A low-power network-on-chip implementing the token flow control router architecture with swing-reduced interconnects. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(8):1432–1446, 2013.

[172] Todd A Proebsting, Gregg Townsend, Patrick Bridges, John H Hartman, Tim Newsham, and Scott A Watterson. Toba: Java for applications — A way ahead of time (WAT) compiler. In *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) — Volume 3*, page 3, Berkeley, CA, USA, 1997. USENIX Association.

[173] James Psota and Anant Agarwal. rMPI: message passing on multicore processors with on-chip interconnect. *High Performance Embedded Architectures and Compilers*, pages 22–37, jan 2008.

[174] Wolfgang Puffitsch and Martin Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, page 221. ACM, 2007.

[175] Wolfgang Puffitsch, Benedikt Huber, and Martin Schoeberl. Worst-case analysis of heap allocations. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 464–478, oct 2010.

[176] Python Software Foundation. Python Programming Language, 2015. URL `https://www.python.org/`.

[177] Python Software Foundation. Built-in Functions — Python 3.4.3 documentation, 2015. URL `https://docs.python.org/3/library/functions.html`.

[178] Hridesh Rajan, Steven M Kautz, Eric Lin, Sarah Kabala, Ganesha Upadhyaya, Yuheng Long, Rex Fernando, and Loránd Szakács. Capsule-oriented Programming. Technical Report 13-01, Iowa State U., Computer Sc., 2013.

[179] D Reinhardt and G Morgan. An embedded hypervisor for safety-relevant automotive E/E-systems. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 189–198, jun 2014. doi: 10.1109/SIES.2014.6871203.

[180] Dominik Reinhardt and Markus Kucera. Domain Controlled Architecture — A New Approach for Large Scale Software Integrated Automotive Systems. In *Third International Conference on Pervasive and Embedded Computing and Communication System — PECCS '13*, pages 221–226, 2013.

[181] Steve Rhoads. Plasma — most MIPS opcodes — OpenCores, 2013. URL `http://opencores.org/project,plasma`.

[182] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, pages 358–366, 1953.

[183] Ronald Rivest. The MD5 message-digest algorithm - RFC1321. *Internet Engineering Task Force — Request for Comments*, 1992.

[184] SAE International. J1939: Serial Control and Communications Heavy Duty Vehicle Network - Top Level Document - SAE International. Technical report, 2013. URL `http://standards.sae.org/j1939{_}201308/`.

[185] Manuel Saldana and Paul Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6. IEEE, 2006. ISBN 1-4244-0312-X. doi: 10.1109/FPL.2006.311233.

[186] M. Norazizi Sham Mohd Sayuti and Leandro Soares Indrusiak. Real-time low-power task mapping in Networks-on-Chip. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 14–19. IEEE, aug 2013. ISBN 978-1-4799-1331-2. doi: 10.1109/ISVLSI.2013.6654616.

[187] M. Schoeberl. JOP : A Java Optimized Processor for Embedded Real-Time Systems. Technical Report 8625440, 2005.

[188] Martin Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(April):176–213, 2010.

[189] Martin Schoeberl, Christopher Brooks, and Edward A Lee. Code generation for embedded Java with Ptolemy. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 155–166. Springer, 2010.

[190] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. Compiling java for low-end embedded systems. *LCTES '03 Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, 38(7):42, jul 2003. ISSN 03621340. doi: 10.1145/780731.780739.

[191] H.R. Simpson. Layered architecture(s): principles and practice in concurrent and distributed systems. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, pages 312–320. IEEE Computer. Soc. Press, 1997. ISBN 0-8186-7889-5. doi: 10.1109/ECBS.1997.581893.

[192] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, jun 1998. ISSN 03600300. doi: 10.1145/280277.280278.

[193] Sriram Srinivasan. Kilim: A server framework with lightweight actors, isolation types and zero-copy messaging. Technical Report UCAM-CL-TR-769, University of Cambridge, Computer Laboratory, feb 2010. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-769.pdf`.

[194] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actor for Java (A Milion Actors, Safe Zero-Copy Communication). In *2008 European Conference on Object-Oriented Programming — ECOOP '08*, volume 5142, pages 104–128–128, 2008. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5.

[195] ST Microelectronics. STM32L152RE Ultra-low-power ARM Cortex-M3 MCU - STMicroelectronics. URL `http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1295/LN962/PF259539`.

[196] Sun Microsystems. Virtual Machine Specification: Java Card Platform, Version 3, Classic Edition. Technical report, 2008.

[197] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. MPI and communication—High-performance and scalable MPI over InfiniBand with reduced memory usage. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06*, page 105, New York, New York, USA, nov 2006. ACM Press. ISBN 0769527000. doi: 10.1145/1188455.1188565.

[198] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[199] D. Sylvester and K. Keutzer. Impact of small process geometries on microarchitectures in systems on a chip. *Proceedings of the IEEE*, 89(4):467–489, apr 2001. ISSN 00189219. doi: 10.1109/5.920579.

[200] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. *Proceedings of the IEEE Symposium on Security and Privacy*, pages 48–62, 2013. ISSN 10816011. doi: 10.1109/SP.2013.13.

[201] Michael Bedford Taylor, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, Anant Agarwal, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, and Jason Kim. Evaluation of the Raw Microprocessor. *ACM SIGARCH Computer Architecture News*, 32(2):2, mar 2004. ISSN 01635964. doi: 10.1145/1028176.1006733.

[202] The Multicore Association. Multicore Communications API Specification V1.063 (MCAPI). http://www.multicore-association.org/workgroup/mcapi.php, mar 2008.

[203] The Open Group. Safety Critical Java Technology Specification. (October), 2010.

[204] T. N. Theis. The future of interconnection technology. *IBM Journal of Research and Development*, 44(3):379–390, may 2000. ISSN 0018-8646. doi: 10.1147/rd.443.0379.

[205] D.E. Thomas, J.K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design & Test of Computers*, 10(3):6–15, 1993. ISSN 07407475. doi: 10.1109/54.232468. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=232468`.

[206] Suleyman Tosun. New heuristic algorithms for energy aware application mapping and routing on mesh-based NoCs. *Journal of Systems Architecture*, 57(1):69–78, jan 2011. ISSN 13837621. doi: 10.1016/j.sysarc.2010.10.001.

[207] Justin L Tripp, Preston A Jackson, and Brad L Hutchings. Sea Cucumber: A Synthesizing Compiler for FPGAs. *Lecture Notes in Computer Science*, 2438(2438): 875–885, aug 2002. doi: 10.1007/3-540-46117-5-90.

[208] Jeff Tsay, Christopher Hylands, and Edward Lee. A code generation framework for Java component-based designs. In *Proceedings of the international conference on Compilers, architectures, and synthesis for embedded systems — CASES '00*, pages 18–25, New York, New York, USA, nov 2000. ACM Press. ISBN 1581133383. doi: 10.1145/354880.354884.

[209] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1937. ISSN 0024-6115. doi: 10.1112/plms/s2-42.1.230.

[210] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. ISSN 00010782. doi: 10.1145/79173.79181.

[211] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P. Rendell. Programming the Adapteva Epiphany 64-Core Network-on-Chip Coprocessor. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 984–992. IEEE, may 2014. ISBN 978-1-4799-4116-2. doi: 10.1109/IPDPSW.2014.112.

[212] Ankush Varma and Shuvra S. Bhattacharyya. Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems. *DATE '04 Proceedings of the conference on Design, automation and test in Europe*, 3:30161, feb 2004.

[213] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, (4):27–75, 1993.

[214] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. Technical report, 1994.

[215] David W Walker and Jack J Dongarra. MPI: A Standard Message Passing Interface. *Supercomputer*, 12:56–68, 1996.

[216] M. Ward and N. C. Audsley. Hardware compilation of sequential ada. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2001.

[217] M. Ward and N.C. Audsley. Hardware implementation of programming languages for real-time. In *Proceedings of the Eighth IEEE Real-Time Embedded Technology and Applications Symposium (RTAS02)*, number figure 1, pages 276–284. IEEE Comput. Soc, 2002. ISBN 0-7695-1739-0. doi: 10.1109/RTTAS.2002.1137403.

[218] Douglas Watt. *Programming XC on XMOS devices*. XMOS Inc., 2009. URL `https://download.xmos.com/XM-001598-PC-1.pdf`.

[219] Andrew Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004. ISBN 047084437X.

[220] David Wentzlaff and Anant Agarwal. Factored operating systems (fos). *ACM SIGOPS Operating Systems Review*, 43(2):76, apr 2009. ISSN 01635980. doi: 10.1145/1531793.1531805.

[221] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE micro*, (5):15–31, 2007.

[222] Lars Wernli. *Design and implementation of a code generator for the CAL actor language*. Diploma thesis, Electronics Research Laboratory, College of Engineering, University of California, 2002.

[223] J.A. Williams, I. Syed, J. Wu, and N.W. Bergmann. A Reconfigurable Cluster-on-Chip Architecture with MPI Communication Layer. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 350–352. IEEE, apr 2006. ISBN 0-7695-2661-6. doi: 10.1109/FCCM.2006.14.

[224] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 362–365. IEEE, dec 2013. ISBN 978-1-4799-2198-0. doi: 10.1109/FPT.2013.6718388.

[225] Niklaus Wirth. Hardware Compilation: Translating Programs into Circuits. *Computer*, 31(6), 1998. ISSN 0018-9162.

[226] Xilinx. Xilinx DS449 LogiCORE IP Fast Simplex Link (FSL) Bus. Technical report, 2010.

[227] Xilinx. AXI Reference Guide UG761 (v14.3). 2012.

[228] Xilinx. MicroBlaze Processor Reference Guide UG081 (v14.2). Technical report, Xilinx Inc., 2012.

[229] Xilinx. UG474 — 7 Series FPGAs Configurable Logic Block. pages 1–72, 2012.

[230] Xilinx. Introduction to FPGA Design with Vivado High-Level Synthesis UG998. 2013.

[231] Xilinx. UG848 — Getting Started with the Virtex-7 FPGA VC707 Evaluation Kit. Technical report, 2014.

[232] Xilinx. DS180 — 7 Series FPGAs Overview. Technical report, 2015.

[233] Yang Zhao, Jie Liu, and Edward A. Lee. A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 259–268. IEEE, apr 2007. ISBN 0-7695-2800-7. doi: 10.1109/RTAS.2007.5.

[234] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam KieÅijun, and Michael D. Ernst. Object and reference immutability using Java generics. *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 75–84, 2007. doi: 10.1145/1287624.1287637.

[235] Christopher Zimmer and Frank Mueller. NoCMsg: Scalable NoC-Based Message Passing. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 186–195. IEEE, may 2014. ISBN 978-1-4799-2784-5. doi: 10.1109/CCGrid.2014.19.