A PROCESS ORIENTED APPROACH TO SOLVING PROBLEMS OF PARALLEL DECOMPOSITION AND DISTRIBUTION

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

BY

DAMIAN J. DIMMICH

JUNE 2009

Abstract

This thesis argues that there is a modern, broad and growing need for programming languages and tools supporting highly concurrent complex systems. It claims traditional approaches based on threads and locks, are non-compositional and do not scale. Instead, it focuses on occam-pi, a derivative from classical Transputer occam whose *process oriented* concurrency model is based on a combination of the formal algebras of Hoare's Communicating Sequential Processes and Milner's pi-calculus. The advent of hybrid processors such as STI's Cell Broadband Engine, which consists of a PowerPC core and eight vector co-processors on a single die, NVidia's graphics-processor based CUDA architecture and Intel's upcoming Larabee require new programming paradigms in order to be used effectively. occam-pi's compositional concurrency model simplifies the management of complexity of concurrent programs and is capable of filling the technological gap that the new processors are creating in terms of the lack of expressiveness of concurrency in current programming languages. occam-pi's formalised basis allows reasoning about programs using formal methods techniques and avoids common concurrency errors though compile-time verification.

The Transterpreter, a new portable runtime for occam-pi reduced the cost of porting occam-pi to new platforms to a minimum. Further extensions to the Transterpreter enable hardware-specific enhancements to the support library, making it possible to implement and evaluate occam-pi on new platforms in a relatively short time.

The work reported in this thesis makes use of this ability and presents implementations of the Transterpreter on new and interesting processors, evaluating the use of process oriented concurrency as a programming model on such processors. Additional infrastructure that is required to make occam-pi useful on such architectures is presented, such as interfacing with legacy languages, thereby providing support for existing libraries and device drivers. Furthermore techniques for making use of vector processing capabilities that are offered by these new architectures are described.

This thesis claims that the work presented makes a useful contribution to simplifying the design and construction of complex systems through the use of concurrency. By enabling both the language and the runtime to support new architectures through libraries, device drivers and direct access to hardware, it enables that contribution for learners and advanced engineers working with novel hardware.

Contents

Abstract ii

Contents iii

List of Tables vii

List of Figures viii

Listings ix

Acknowledgements xi

1 Introduction 1
- 1.1 A Portable Runtime for occam-pi . . . 2
- 1.2 Controlling C . . . 3
- 1.3 The Cell Broadband Engine and occam-pi . . . 3
- 1.4 Contributions . . . 4
- 1.5 Publications . . . 5
- 1.6 Organisation . . . 6

2 Background 7
- 2.1 Key occam Concepts . . . 9
  - 2.1.1 Going Parallel . . . 9
  - 2.1.2 Channel Communication . . . 10
  - 2.1.3 Introducing non-determinism . . . 11
  - 2.1.4 A simple occam program . . . 11
  - 2.1.5 occam and the Transputer . . . 12
- 2.2 Introducing occam-pi . . . 13
  - 2.2.1 Dynamic memory and dynamic replication . . . 15
  - 2.2.2 User Defined Operators . . . 16
  - 2.2.3 Extended Rendezvous . . . 16
  - 2.2.4 Clustering workstations in occam-pi . . . 17
- 2.3 The Anatomy of an occam-pi Program . . . 18
  - 2.3.1 Runtimes . . . 18
  - 2.3.2 CCSP . . . 19
  - 2.3.3 The Transterpreter . . . 19

2.3.4 Summary of Runtimes . . . 22
2.4 Compilers . . . 22
2.4.1 OCC21 . . . 22
2.4.2 Tock . . . 23
2.5 Motivating occam-pi usage today . . . 24

3 occam-pi as a control language **26**
3.1 Background and motivation . . . 27
3.2 Integrating libraries into occam-pi . . . 28
3.3 Interface Generators . . . 30
3.4 SWIG Module Internals . . . 31
3.5 An Overview of the occam-pi FFI . . . 32
3.6 SWIG and the occam-pi FFI . . . 34
3.6.1 Using SWIG to Generate Wrappers . . . 34
3.6.2 Generating Valid occam-pi PROC Names . . . 34
3.6.3 Auto-generating Missing Parameter Names . . . 34
3.6.4 Data Type Mappings . . . 35
3.6.5 Structures . . . 35
3.6.6 Unions . . . 37
3.6.7 Pointers and Arrays . . . 38
3.6.8 Typeless pointers . . . 38
3.6.9 Enumerations . . . 40
3.6.10 Preprocessor directives . . . 41
3.6.11 Wrapping Global Variables . . . 42
3.6.12 Non-blocking C calls . . . 42
3.6.13 Wrapper Documentation . . . 43
3.7 Using SWIG . . . 43
3.8 Examples of automatic library wrapping . . . 46
3.8.1 A Simple Math Library Demo . . . 46
3.8.2 Wrapping OpenGL . . . 48
3.9 Possible pitfalls when using C with occam-pi . . . 50
3.9.1 Ensuring ordering of foreign function calls . . . 50
3.9.2 Mapping external events to occam-pi channels . . . 52
3.9.3 Working with libraries that rely on callbacks . . . 53
3.10 Chapter contributions and summary . . . 56
3.10.1 Projects that made use of SWIG occam-pi . . . 57
3.10.2 Future Improvements to the SWIG Module . . . 58
3.10.3 SWIG and CIF . . . 58

4 occam-pi on the Cell BE **59**
4.1 An Overview of the Cell Broadband Engine . . . 60
4.1.1 PowerPC Core (PPC) . . . 60
4.1.2 Synergistic Processing Units (SPU) . . . 60
4.1.3 The Cell's Element Interconnect Bus . . . 62
4.2 Current approaches to programming the Cell . . . 62

4.2.1 OpenMP . . . 64
4.2.2 The Octopiler . . . 67
4.2.3 Stream Programming for the Cell . . . 68
4.2.4 MPI Microtasks for the Cell . . . 70
4.2.5 Trancell — an occam-pi translator for the Cell . . . 71
4.2.6 CellCSP . . . 71
4.2.7 Bandwidth/Optimisation Techniques on the Cell . . . 72
4.3 The Transterpreter on the Cell Broadband Engine . . . 73
4.3.1 Program Distribution . . . 73
4.3.2 Inter-Processor Communication . . . 74
4.3.3 Communication using specialised registers . . . 74
4.3.4 Using DMA for communication . . . 76
4.3.5 Scheduling . . . 78
4.3.6 Timers on the Cell . . . 79
4.4 The Cell Transterpreter API . . . 79
4.4.1 Determining execution locality . . . 80
4.4.2 The Channel API . . . 80
4.4.3 Using Mailboxes . . . 82
4.5 occam-pi channel communication for the Cell . . . 84
4.5.1 A microkernel for channel control . . . 85
4.5.2 Limitations of the current implementation . . . 89
4.6 An evaluation of Mandelbrot algorithms for the Cell . . . 90
4.6.1 Overview . . . 90
4.6.2 A Simple Implementation . . . 91
4.6.3 Making use of explicit concurrency . . . 93
4.6.4 Decoupling work units from SPUs . . . 96
4.7 Further Work on the Cell Transterpreter . . . 99
4.7.1 Arbitrary/dynamic channel networks . . . 99
4.7.2 Vector processing . . . 99
4.7.3 Code Generation . . . 100
4.7.4 Efficient interpretation on the Cell . . . 101
4.8 Summary . . . 101

5 Vector Processing in occam-pi 102
5.1 Vector processing background . . . 102
5.2 How vector processing is currently used . . . 103
5.3 Extensions to GCC for vector processing . . . 104
5.4 Stream programming . . . 105
5.5 Vector processing in occam-pi . . . 106
5.5.1 Using GCC vector intrinsics from occam-pi . . . 106
5.5.2 Performing more computations in C . . . 110
5.5.3 Operator Overloading and Vectors . . . 112
5.6 Vector support for current runtimes . . . 114
5.6.1 Automatic vectorisation in occam . . . 115
5.7 Benchmarks . . . 116

5.8 Conclusions and future work . . . 117

6 Further contributions **119**
6.1 occam-pi and Player/Stage . . . 119
6.2 The 42 compiler . . . 121
6.3 Extensions to the Transterpreter . . . 124
6.4 occam-pi on other platforms . . . 126
6.4.1 16-bit DOS . . . 126
6.4.2 Symbian S60 . . . 126
6.4.3 Gameboy Advance . . . 127
6.5 Summary . . . 128

7 Conclusion **129**
7.1 Contributions . . . 130
7.2 Short-term future work . . . 131
7.2.1 Automatically generating CIF for Callback functions . . . 131
7.2.2 Support for C++ from occam-pi . . . 131
7.2.3 Standardised graphical interface . . . 132
7.2.4 Transterpreter Cell enhancements . . . 132
7.2.5 General Transterpreter enhancements . . . 134
7.2.6 A more exhaustive comparison of algorithms . . . 136
7.3 Long-term future work . . . 136
7.3.1 Support for CUDA/GPU/Larabee-like architectures . . . 136
7.3.2 Tock support for the Transterpreter as a back-end . . . 137
7.3.3 Compiler support for generating code for the Cell . . . 137
7.3.4 Compiler support for SIMD processing . . . 138

Bibliography **139**

List of Tables

1 Standard occam-pi operators. 16
2 Extended occam-pi operators for user defined operations. 16
3 An overview of the occam-pi runtimes. 22
4 GL/gl.h and GL/glu.h interface files for occam-pi 49
5 Sample results for vector processing in occam-pi. 118

List of Figures

1 An example network of Transputers. 13
2 A graphical representation of the occam-pi compilation process. 18

3 Lazy simulation of Conway's Game of Life. 50
4 Interacting with events using occam-pi and SDL. Events are taken off the queue if available and sent to the remainder of the application. . . . 53
5 Interacting with callbacks using occam-pi. The status variable must either be polled from occam-pi or the callback function must be instrumented to use CIF. 56

6 A diagram of an SPU. 62
7 A diagram of the Cell BE. 63
8 A model of how concurrency is achieved using OpenMP. 64
9 The read/write buffers allow computation to continue with less interruption. 78
10 The layout of a channel's data in memory. 82
11 A visual representation of the steps taken during channel communication. 88
12 Parallel Mandelbrot on the Cell. 90
13 A Mandelbrot set rendered on the Cell. 96

14 Two Pioneer robots in the Gazeebo virtual environment controlled from occam-pi. 120

Listings

2.1 Mixing sequential and parallel. 10
2.2 Channel communication resulting in Deadlock. 10
2.3 Channel communication not resulting in Deadlock. 11
2.4 A "Hello world" program in occam-pi 12
2.5 An sample process demonstrating extended rendezvous. 17
3.1 A C function prototype 32
3.2 Converting occam-pi pointers for use in C 32
3.3 Calling C from occam-pi 33
3.4 glCallLists original prototype. 38
3.5 glCallLists prototype modified for SWIG-occam-pi. 38
3.6 An example of `glCallLists` used to render text in occam-pi. 39
3.7 Data-type aware wrappers for functions that take **`void`** parameters. . . . 40
3.8 An example C **`enum`**. 40
3.9 Code generated by SWIG for an **`enum`**. 40
3.10 Sample preprocessor directives in C. 41
3.11 Preprocessor directives translated to occam-pi. 41
3.12 Example C header file calc.h 46
3.13 Example SWIG interface file `calc.i`. 47
3.14 Example program using SWIG generated code. 47
3.15 A build script for swig-occam-pi. 48
3.16 OpenGL interface file for SWIG. 49
3.17 A section of OpenGL code that needs to be executed serially. 51
3.18 An event monitor using SWIG and occam-pi 54
4.1 OpenMP fork-join parallel example 65
4.2 Race hazard example using OpenMP. 66
4.3 Race condition free OpenMP example. 66
4.4 A sample factorial function in CellSs taken from [PBBL07]. 68
4.5 Invoking a CellSs kernel. 69
4.6 An example of a start up process on the Cell Transterpreter. 80
4.7 The channel API 81
4.8 A process running on an SPU. 83
4.9 The process which runs on the PPC and outputs data to the screen. . . . 83
4.10 A startup process which starts other processes depending on program location. 84
4.11 A reading process using DMA. 85

4.12 A writing process using DMA. 87
4.13 OpenMP implementation of Mandelbrot pseudo code. 91
4.14 Simple server for a Mandelbrot set generator in occam-pi. 92
4.15 Client end of a simple Mandelbrot set generator in occam-pi. 93
4.16 Server end of an efficient implementation of a Mandelbrot set generator in occam-pi. 94
4.17 Client end of a Mandelbrot set generator with multiple workers. 95
4.18 Server end of a farming implementation of a Mandelbrot set generator in occam-pi. 96
4.19 Client end of a farming implementation of a Mandelbrot set generator in occam-pi. 98
5.1 Using GCC built-ins for vector processing 104
5.2 Brook vector programming example 105
5.3 Creating two vectors from scalar values and allocating a third, emtpy vector. 107
5.4 Creating a vector in C. 108
5.5 Multiplying vectors in occam. 109
5.6 Vector multiplication helper function. 109
5.7 Freeing previously allocated vectors from occam 109
5.8 C helper functions for freeing occam vectors. 110
5.9 A function that allocates an array of vectors and populates it with data. . 110
5.10 A function that multiplies two lists of arrays and stores the result in a third array. 111
5.11 Frees an array of vectors. 111
5.12 User defined operator for arrays of vectors. 114
5.13 A **PAR** that could benefit from automated vectorisation. 115
5.14 Vector calculation benchmark pseudo code. 116
5.15 Allocate a vector in C from four values. 116
5.16 Allocate a vector in C from an occam array. 117
6.1 A sample S-expression occam program. 123

Acknowledgements

In particular I would like to thank Christian Jacobsen for first suggesting the idea of a research degree, helping me through it and teaching me much of what I know about programming. This thesis is built on many of the technologies that Christian and Matt Jadud and the concurrency research group developed. A special thanks also goes to my supervisor, Peter Welch for agreeing to take me on as a student and always providing invaluable support and advice throughout the thesis.

I would also like to thank:

Members of my research group in no particular order, Adam Sampson (for rocking out), Fred Barnes, Jon Simpson[1], Matt Jadud, Carl Ritson and Neil Brown for always being available for feedback, criticism and to discuss new ideas with. It has been a pleasure working with all of you.

Magdalena Stepien, Richard, Lena, Kamil Dimmich and Bianca Bernardo for being there when I needed them and for supporting me throughout the PhD. In particular Magdalena for her patience and understanding while I continued to promise that I would move in with her for over a year.

My friends many of whom have undertaken similar endeavours, also in no particular order, Poul Henriksen, Edward Suvanphen, Rick Walker, Tara Puri, Benoit Dillet, Srivas Chennu, Patrick Craston, Pulitha Liyanagama, Sebastian Marion, Kat Johnson, Federico Gallo, Michael Pediaditakis[2], Keith Franklin, Nick Miles, Sabrina Paneels, Akiko Uri, Axel Simon, Rodolfo Gomez, Gift Nuka, Brad Wyble, Annette Kuhn, Alvin Du, Philip Ganev, James Munro and my cousin Krystian Soroko. Thank you for making my time in Canterbury as good as it's been. I hope all our paths cross again.

Finally, my thanks goes to the Computing Laboratory and their support staff thanks to whom this thesis was made possible, both in terms of funding for two years and supporting the continued development of this thesis when I needed it most.

[1] Jon for allowing me to steal his bouncy medicine ball.

[2] Thanks also goes to Mike for introducing me to MegaRock Radio, under whose influence much of this thesis was written.

Chapter 1

Introduction

This thesis demonstrates that the occam-pi programming language provides a set of abstractions that are well suited to modern parallel processors and provide a path for the parallelisation of existing programs. This is achieved using the Cell Broadband Engine and the Transterpreter as its primary case study.

occam-pi [BW04] builds on the formal algebras of CSP [Hoa85] and the pi-calculus [Mil99, MPW92] to create a novel concurrent programming language. The language encourages concurrency as a design methodology — process oriented programming — by providing explicit support for concurrency as part of the language's syntax and semantics. occam-pi is a superset of the occam2.1 language [INM84c] and continues the tradition of providing safe, composeable and lightweight concurrency for programmers. occam2.1, initially designed for use on the Transputern [INM88, Wik07] and later adapted for other processors [DHWN94, Bar01, WW96], was used in numerous applications ranging from embedded control to large computationally intensive application [RWW91, DBB+93].

Previously occam-pi has focused on the use of the language on desktop-class computers and commodity clusters and supported only Intel platforms resulting in a narrower application area for occam-pi. To broaden application support the Transterpreter runtime and virtual machine [JJ04, JJ05] for occam-pi was created, which enabled the language to be used, for example, on embedded platforms with limited amounts of memory [JJ05, SJJ06]. The Transterpreter is written in C and relies on the general

availability of C compilers for most platforms. This enables the occam-pi developer to concentrate on exploring applications of the language on new and interesting platforms by simplifying the process of porting the runtime.

1.1 A Portable Runtime for occam-pi

One of the original motivating platforms for which the Transterpreter was developed is the LEGO Mindstorms RCX [Gro04]. The RCX is an inexpensive robotics set, consisting of a 16-bit microprocessor with 32KB of RAM and three sensor inputs and three motor outputs, providing the facilities required to build small robots. This platform was chosen as it provides a compelling vehicle for the teaching of concurrency using the occam2.1 programming language. Even though the Transterpreter's design was motivated by a small embedded device — the RCX — it also provided a means to run occam-pi on previously unsupported platforms. On a 16-bit platform, such as the RCX, the Transterpreter binary is around 11KB in size and 6KB of the total 32KB being used by the RCX firmware, leaving approximately 15KB of RAM available for program byte code and memory required for execution of the program. This opens new application areas for the occam-pi language, including sensor networks and small-scale robotics.

The Transterpreter executes a concise byte code based on the Extended Transputer Code (ETC) instruction set, a superset of the original Transputer instruction set, first described by Michael Poole in [Poo98]. OCC21 — the occam compiler — generates ETC as an intermediate step in the compilation process. Previously ETC was translated to a processor's native assembly by tranx86 and linked with the CCSP runtime library which provided the necessary environment for an occam-pi program execution. The Transterpreter replaces tranx86 and CCSP in the compilation process, and due to its design, allows for relatively quick retargetting of occam-pi to new platforms. This thesis explores usage of the Transterpreter on new platforms where the availability of occam-pi can prove to be a compelling alternative compared to the currently available languages.

1.2 Controlling C

While the Transterpreter simplifies occam-pi's cross-platform portability, and makes the language available on embedded devices, giving it a broader range of uses, it does not facilitate the re-use of existing software libraries. The ability to re-use software is key a key factor ensuring that a language is both useful and extensible. occam-pi has support for interfacing with such libraries; however the process of creating such an interface requires a large amount of manual effort. To facilitate the reuse of existing software and decrease the amount of effort required to interface with new libraries and hardware, a less labour intensive approach is needed. An automated approach is also invaluable for the maintenance of interfaces to evolving libraries and APIs.

A popular library interface generator framework — SWIG — [Bea96] has been extended to support occam-pi as part of this thesis. This broadens the scope of applications for the occam-pi language enabling it to be used as a control language for existing libraries as well as interfacing with new platforms. The occam-pi wrapper generator provides a reusable mechanism for interfacing with such libraries, allowing a programmer to migrate existing code to be used with a safe and robust concurrent language. This supports the motivations for this thesis by promoting reuse of existing libraries and code to bootstrap and speed development of occam-pi on new and existing platforms.

1.3 The Cell Broadband Engine and occam-pi

As the primary case study for parallelising existing software, occam-pi is used as a control language on the Cell Broadband Engine. The Cell Broadband Engine [Pha05, KDH+05] (which will be referred to as the CBE or Cell), a processor developed by Sony, Toshiba and IBM, is a good example of a processor that benefits from a language that supports concurrency. It was developed for multimedia/streaming purposes focusing on high levels of vector processing capability and high sustained memory bandwidth. At its core lies a Power5 [RSJ04] (PPC) based processor which is combined

with eight SPUs [Fla05, Mue05] — dedicated vector processors that each have 256KB programmable caches, all on a single die. The SPUs, the PPC processor and the system memory controller are interconnected by a high speed on-chip ring bus, known as the Element Interconnect Bus (EIB) [KPP06].

The Cell is used in applications that are computationally demanding such as multimedia, games [Ent06] and scientific computation [WSO+06]. It is available in high-end blade servers and also found in the widely available Playstation III games console. At the Cell's launch a common criticism by the developer community was that it would be difficult to program because of its complexity [SVP07]. The cited reasons were difficulty of partitioning work across the processor, having to deal with 9 distributed memory spaces and their synchronisation, and having to deal with two different instruction sets on a single chip as the SPU processors have their own ISA [Sha06].

The Cell provides a compelling platform to use as a case study for the occam-pi language and the Transterpreter runtime. The runtime is small enough to be able to execute in the limited 256KB memory of the SPU processors while still providing full support for the occam-pi language. When adapted for the Cell the Transterpreter is able to provide language-level support for safe synchronisation and copying of data between processes and processors which would help in addressing some of the issues identified above.

1.4 Contributions

The central contribution presented in this thesis is an evaluation of a process oriented programming approach to solving problems of parallel decomposition and distribution on the Cell Broadband Engine. An implementation of the Transterpreter for the Cell is described and programming examples are given. These are then compared to current options for programming the Cell and their relative merits are described in Chapter 4. To support the thesis a number of further contributions which build on the ideas of using the Transterpreter and occam-pi as a control language are presented in Chapter

3, where the automated C interface generator is described. Finally, following the motivation to expand the availability and utility of occam-pi, the development of support for hardware based vector processing as well as ports of the Transterpreter to other platforms are described Chapter 6. Further minor contributions discussed in this thesis include:

- A proposal for an extension to the languages syntax to support vector processing implicitly.
- Extensions to the Transterpreter runtime to support faster floating point calculations and adding support for newer occam-pi language features including some of the mobile pi-calculus features and barriers.
- An experimental multi-pass[1] occam-pi compiler, "42", for the purpose of rapid prototyping of new language features. The ideas developed in this compiler led to the development of the Tock compiler which expanded and improved on "42"s achievements.
- A port to the Transterpreter to the ARM based Nokia S60 and Gameboy Advance platforms as candidates for robotic controllers using occam-pi. An evaluation of this work is provided.

1.5 Publications

The authors key contributions are described in a number of publications which are referred to throughout this thesis. The work on automated foreign function interfaces is described in [DJ05]. The publication [DJJ06] describes the authors initial work on the Cell BE. [JDC06] describes the work on the "42" compiler and the results of the experiment. A further publication to which the author contributed is [JJD06] which describes the use of the Transterpreter in sensor networks. Additional contributions

[1] Each transformation of the parse tree is a separate pass

where made in [JJ07] which describes an environment for teaching concurrency and paralellism through robotics simulation.

1.6 Organisation

Chapter 2 provides background information about the occam-pi language, its development history, the current state of the tools and environments surrounding the language. It also gives details about the Transterpreter's implementation. The implications of using occam-pi on a distributed memory system are discussed and some background about commonly used methods for programming parallel and distributed memory systems are presented. Chapter 3 describes the SWIG-occam-pi module for automatically generating C bindings for occam-pi and its use. Chapter 4 details the internals of the Cell Transterpreter, its use and how it compares to other methods of programming the Cell. Chapter 5 describes a cross-platform vector processing library and other findings that arise as a result of the work performed for this thesis. Other contributions such as the "42" compiler, occam-pi on robotics platforms as well as other experiments conducted during this thesis are described in Chapter 6. Finally, conclusions and future work are described in Chapter 7.

Chapter 2

Background

The occam family of languages are based on the Communicating Sequential Process [Hoa85] calculus (CSP), a formal algebra that can be used to describe concurrent, communicating processes, and allows one to reason about them mathematical. occam processes, like CSP processes, can be run either sequentially or in parallel. When run in parallel processes must communicate over well defined channel interfaces. The semantics of the occam languages permit compile-time analysis of programs which detect and prohibit many common problems faced in concurrent programming.

The occam1 [INM84b] programming language was first released in 1983. An updated version of the language occam2 [INM84c] was released in 1986, followed by occam2.1 [INM95] which was released in 1988. occam2.1 became the de-facto standard way of writing occam programs and was the preferred version of the language, containing many features expected in a modern language at the time that occam1 did not have. occam-pi [BW02], first introduced in 2001, is a source-compatible successor to occam2.1. occam-pi underwent a number of extensions and had new features added over time. It is the most modern variant of occam, and as such is the primary language used in this thesis.

Originally occam was compiled for the Transputer, a processor developed in tandem with the occam language. As the availability of the Transputer processors diminished in the 90's, occam2.1 was retargetted at processors such as the PowerPC, Sparc, IA32, SHARC DSP and Alpha [SARW98, WW96, Bar01, BOS+96, WP97] through

the KRoC compiler, OCC21, created as part of the occam-for-all (OFA) project. Two approaches to retargetting occam existed. The first approach modified OCC21 to directly generate binary code for each new target architecture. The second and currently used approach, first proposed in [Poo96], created a version of OCC21 which outputs an intermediary representation, which is translated to binary using a separate tool. The intermediary representation initially used was Transputer assembly, and later an extended transputer assembly (ETC) [Poo98] as the compiler evolved. ETC in turn would be translated to native assembly by an architecture-specific translator then no longer requiring modification of the compiler and significantly reducing the time it takes to port the compiler to a new architecture. Shortly after the release of the first Transputers in 1984, occam1 was made available through the Portakit [INM84a, Bra85]. This occam1 interpreter ran on a variety of systems, however it was said to contain errors in the occam model and was not compatible with future versions of the language. After the demise of the Transputer in the early 1990's a Transputer emulator [Hig97] was created for the purpose of running native transputer binaries which allowed for the execution of occam1 and occam2.1 programs. More recently however OCC21, evolved to compile the occam-pi language which contains many features not supported on the original Transputers. OCC21 only targets the extended transputer assembly (ETC) which is not binary compatible with the original instruction set. Another compiler, SPoC [DHWN94], took a different approach and generated target independent C code instead of translating occam2.1 programs into native assembly. The SPoC compiler has not been actively maintained and is not able to compile the newer occam-pi language.

More recently the Transterpreter, a virtual machine, enables almost direct execution of the ETC that Poole's OCC21 produces leading to a greater degree of hardware independence for both the occam and occam-pi languages. This enables the language to be used and evaluated on new types of hardware/platforms avoiding the need to write a new translator, further reducing the time and effort required for porting. In particular, at the time of the Transterpreter's inception, all but the support for the IA32 platform had fallen into disuse, due to the overhead required in maintaining the native compiler

translators.

In particular, the Transterpreter allowed occam-pi to be used for educational purposes on platforms such as the LEGO Mindstorms a small robotics platform that provides a compelling environment on which students can explore and learn about concurrency. Teaching concurrency on an engaging platform like the LEGO Mindstorms was one of the primary motivations for creating the Transterpreter.

2.1 Key occam Concepts

This secton presents key occam syntax, and is inteded to give an overview of the occam languages features before going on to explain how the Transputer hardware and occam are related. For a more in depth guide to the occam2.1 language please refer to the freely available Jones' "Programming in occam 2" [JG88] or Burns' "Introduction to the Programming Language Occam" [Hyd95], both of which are relevant to occam-pi as it shares and extends occam2.1's syntax.

2.1.1 Going Parallel

Everything in occam programs is a process. occam syntax uses the **PAR**allel and **SEQ**uential keywords to denote whether a list of processes should run in parallel or in sequence. Implicit barriers at the end of **PAR** blocks ensure that parallel processes are synchronised upon completion, which on their own result in entirely deterministic programs because updates on shared data are disallowed. For example, in Listing 2.1, `process1` executes first and on completion, `process2` and `process3` execute in parallel, only continuing on to `process4` once both processes complete. Starting multiple processes in occam has very low start-up and shut down costs and, when running, the cost of a context switch is very low. In stark contrast to operating system processes or even threads, occam processes are extremely light-weight and use co-operative scheduling.

```
SEQ
  process1
  PAR
    process2
    process3
  process4
```

Listing 2.1: Mixing sequential and parallel.

2.1.2 Channel Communication

occam channel communication is unidirectional, blocking and point to point. Because of these properties each communication is also a synchronisation point, and the system is in a known state. Non-determinism must be introduced explicitly using a specific syntax. These properties, based on the rules of the CSP calculus, are what enable formal reasoning about concurrent systems created with occam. They allow the compiler to check for many common concurrency errors at compile time, and permit run-time deadlock detection. For example, Listing 2.2 shows a deadlock scenario which would be detected at compile time, and the corrected version in Listing 2.3 which would pass the compilers checks. Syntactically channels in occam are defined with the **CHAN** keyword, and communication on channels is denoted using the send `!` and receive `?` symbols. Processes wishing to communicate through channels must be run in parallel, otherwise deadlock would occur because of the blocking nature of channels.

```
CHAN BYTE a:
BYTE var:
SEQ
  a ! 'a'
  a ? var
```

Listing 2.2: Channel communication resulting in Deadlock.

```
CHAN BYTE a:
BYTE var:
PAR
  a ! 'a'
  a ? var
```

Listing 2.3: Channel communication not resulting in Deadlock.

2.1.3 Introducing non-determinism

The **ALT**ernate construct is used to introduce non-determinism in to parallel programs. The **ALT** keyword allows the arbitrary selection of input from a set of channels, and hence allows for explicit non-determinism to be introduced into the language. This is useful as it allows a process to monitor multiple channels simulataneously, and is desirable to design complex systems of processes.

2.1.4 A simple occam program

In the occam languages syntax scope is determined by indentation. A program will typically start with a single top level process, with keyboard (keyboard input), screen (stdout) and error (stderr) channels channels which provide the interface to a terminal on the users computer. These are special channels in that they are runtime dependent and can be specified arbitrarily by the languages implementation to suit the target architecture. An occam program running on a non-desktop device may not need console I/O, and could therefore have a different default set of channels available to it for interfacing with the system. The starting process is, by convention determined by being the last process specified in the compiled file and is not specified by name as in many other languages. Listing 2.4 shows a "Hello world" program in occam-pi. This program demonstrates, **PROC**ess, **CHAN**nel and **VAL ... IS ...** (constant) declaration as well as **SEQ**ential replication and channel communication. The program begins by declaring an array of characters, i.e. a string constant. It then iterates over all of the values in the array and sends them through the `scr` channel using the write — `!` —

operator, printing each character to the terminal.

```
PROC hello(CHAN BYTE kyb?, scr!, err!)
  VAL BYTE str IS 'Hello world*n':
  SEQ i = 0 FOR SIZE str
    scr ! str[i]
:
```

Listing 2.4: A "Hello world" program in occam-pi

2.1.5 occam and the Transputer

The design of the occam language and the inception of the Transputer processor family are closely related as the Transputer processors were designed with hardware support for executing and scheduling very lightweight concurrent processes. This permitted very fast context switching and supported occam's parallel and channel communication constructs.

Transputers also had the ability to interconnect with other Transputers to form networks of Transputers enabling true parallelism as shown in Figure 1. To support this, each Transputer had four unidirectional high speed links which where used as processor interconnects, and would allow the creation of networks of arbitrary topologies, within the constraints of the available links. The occam language was designed to support networks of Transputers such that processes, or groups of processes could be statically assigned to individual processors. Processes would communicate with each other through occam channels which could either be mapped in memory for intra-processor communication or, bound to the hardware links allowing both interprocess and interprocessor communication using the same communication syntax.

Transputers could be configured through software to act as routers to enable any given processor/process to communicate with any other processor/processes. The latest version of the Transputer, the T9000, also supported connectivity to other processors through a hardware router, allowing arbitrary inter-processor communication. The T9000 also permitted multiple channels to be mapped onto a single link and the runtime was able to multiplex these automatically.

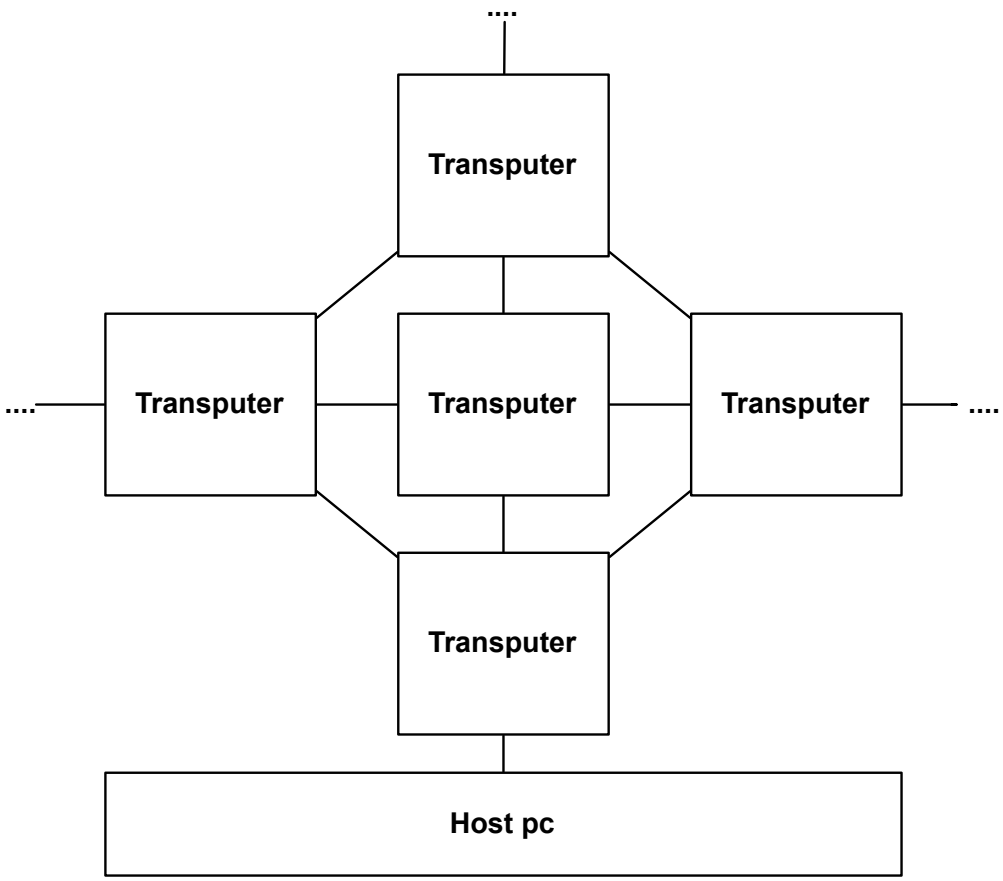

Figure 1: An example network of Transputers.

Each Transputer had its own memory in which to store data and programs, and all communication, including bootstrapping, to the outside world occurred through the high speed links. The nature of the links, and the Transputers main application area, embedded computing with limited memory, meant that occam had to be a very static language, with predictable memory and interconnectivity requirements. The interoperation of the occam language and the Transputer are relevant as their hardware-software co-design background forms the basis of the ideas discussed in Chapter 4.

2.2 Introducing occam-pi

occam-pi's development was motivated by the desire to build large computational

models, consisting of millions of fine grained processes. occam-pi introduces a number of features making the language more dynamic and modern and targets current general purpose processors where the original memory and interconnect constraints of the Transputer no longer apply. The addition of mobility features such as dynamic memory allocation, mobile data, mobile channels, process forking and parallel recursion to occam-pi, stemming from the Pi-Calculus [MPW92], allow for the creation of large dynamic networks in contrast to the purely static networks possible in occam2.1. Other extensions to the language include support for user defined operators and a number of new synchronisation primitives such as shared channels, extended rendezvous and barriers and their mobile equivalent, mobile barriers. These developments are described in detail in a series of papers mainly by Barnes and Welch [WB05, BWS05, BW04, BW02, BW03a, WB01, WM99] and in Barnes's thesis [Bar03a].

The latest versions of occam-pi on KRoC execute on IA32 compatible single and multi-core systems, allowing one to write programs that scale with the availability of additional cores. This has redefined the way occam-pi programs are written, as one does not need to consider the network of processors and their interconnects when designing programs, merely splitting the tasks into small enough chunks to allow them to execute concurrently without requiring that the programmer specify the location of execution. When clusters of such machines are used however, programs must be written with some thought regarding the underlying architecture, as interconnect bandwidth and data locality need to be considered.

The occam-pi compiler goes to great lengths to help the programmer in designing correct concurrent programs, not only by performing syntax checks but also by attempting to enforce necessary parallel safety rules at compile time. The compiler enforces strict aliasing rules, disallows parallel use of variables and channels, enforces directional channel communication, as well as performing type checks and warning when undefined variables are used. Issues of deadlock, live-lock and starvation are not addressed by the compiler, but the runtimes are able to detect deadlock at execution time. Deadlock, live-lock and starvation can be addressed by using certain design patterns [JW92] when writing occam-pi programs or by model checking them using a tool

such as FDR [For00].

The following sections outline the language features in occam-pi relevant to this thesis.

2.2.1 Dynamic memory and dynamic replication

The addition of dynamic memory support to the occam-pi language has allowed the language to become useful on more modern systems with larger amounts of available memory. The **FORK** keyword for example allows new processes to be spawned at run-time. This is useful in scenarios such as servers, where new processes can be spawned and terminated dynamically to deal with incoming requests. **FORK**ed processes only allow shared channels, mobile channels, mobile data or constants (**VAL**) as parameters in order to prevent parallel access to resources, as it is not possible to determine correct parallel usage of data passed to forked processes at compile time.

occam-pi's **MOBILE** data types permit variables to be allocated dynamically at run-time. **MOBILE** data provides flexibility in program design as well as performance increases. Using mobiles, it is possible to pass references where data would previously have been copied, reducing the amount of copying required, for example, when sending arrays through channels. occam-pi's strict aliasing rules ensure safe parallel usage — references to mobile data are lost by the originating process when **MOBILE** data are passed as a parameter to a **FORK**ed process or sent across a channel. In effect the programmer is unaware that they are dealing with pointers and are not exposed to any of the dangers associated with them in other languages.

Furthermore, features such as process recursion, dynamic (run-time determined) replication counts in **PAR** blocks and **MOBILE** channels — where channel ends may be passed between processes through channels — allow occam-pi more flexibility in program design than its static precursor occam2.1.

Binary	+ − /\ \/	* / \ ><	AND OR	PLUS MINUS	TIMES AFTER	<= >= = <>
Unary	−	MINUS	~	NOT		

Table 1: Standard occam-pi operators.

Binary and Unary							
??	@@	$$	%	%%	&&	<% %>	<& &>
<] [>	<@ @>	@	++	!!	==	^	

Table 2: Extended occam-pi operators for user defined operations.

2.2.2 User Defined Operators

User defined operators allow occam-pi's standard operators to be redefined to work with user made data-types, such as `STRUCT`ures. Tables 1 and 2 show all the operators available to a developer for use in user defined operators. The extended operators were added to supplement the standard set of operators should they be required for more complex data types defined by the user program or a library. User defined operators are discussed further in Chapter 5.

2.2.3 Extended Rendezvous

The extended rendezvous allows a reading process control of when it releases the sending process by extending the completion of channel communication until it is ready. During normal communication both processes are made runnable once they have synchronised and exchanged data. While this is sufficient in most cases, there are synchronisation patters that can be difficult to implement using normal communications. For example, the extended rendezvous allows a listening process to be inserted between two processes for monitoring purposes without changing the semantics of the program. An example of a logging process using extended rendezvous is shown in Listing 2.5. While the same program using normal communications would have introduced buffering — thereby changing the behaviour of the channel into which it was inserted, using the

extended rendezvous prevents the buffering from occurring by deferring the release of the `in` channel until the following process completes (in this example, the **SEQ** blocks following any input). This design pattern is useful for debugging purposes or logging of data in a running system, as well as other situations and is illustrated further in Chapter 4.

```
PROC log(CHAN INT in?, out!, VAL []BYTE filename):
  WHILE TRUE
    INT input:
    in ?? input
      SEQ
        write.to.disk(filename, input)
        out ! input
:
```

Listing 2.5: An sample process demonstrating extended rendezvous.

2.2.4 Clustering workstations in occam-pi

Pony [SBW03] is a clustering system that was developed for occam-pi as part of the KRoC tool-chain. Pony builds on many of the ideas that were present in the T9000 Transputer. A cluster, composed of x86 nodes connected by ethernet, is essentially transformed into a network of occam-pi "transputers". An advantage of such an approach is that each node is able to communicate with every other node directly, overcoming the Transputer's restriction of four links. While this permits arbitrary topologies, the physical interconnect performance will vary depending on the actual topology of the networks and needs to be taken into account if performance is important.

In order to support arbitrary channel communication between nodes, a server node provides an inter-node channel creation and routing service. All nodes connect to this server and can request channel connections to other nodes from it. When the server receives such a request, it creates a mobile channel and returns the channel ends to the two nodes that wish to communicate. Once each of the nodes has received its end of

the mobile channel, they are able to communicate directly using occam-pi's channel communication semantics. Once a network channel is set up, the channel's end points can be passed to other nodes, allowing the network of occam-pi nodes to be adjusted dynamically.

2.3 The Anatomy of an occam-pi Program

The occam-pi language's tool chain currently has two runtimes and three compilers. While other compilers for occam2.1 exist, they are no longer of interest to this thesis as they do not support the new occam-pi language. This section gives an overview of the currently available runtimes and compilers. A diagram depicting the possible compilation processes of an occam-pi program can be see in Figure 2.

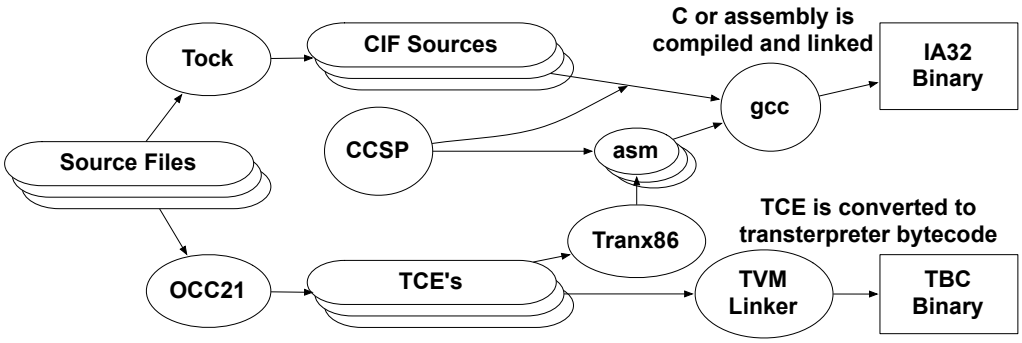

Figure 2: A graphical representation of the occam-pi compilation process.

2.3.1 Runtimes

The two runtimes currently available for occam-pi are CCSP [Moo99] and the Transterpreter [JJ04], which is described in detail in Jacobsen's thesis [Jac08]. The runtimes for occam-pi provide the execution environments that occam programs require such as scheduling and memory allocation. The runtimes contain some features commonly found in operating-systems and have been used as a basis for the formation of small concurrent operating-systems and operating-environments [BJV03, SJJ07].

2.3.2 CCSP

CCSP is a runtime for occam-pi written using a mix of C and assembly, currently only available for IA32 based machines. The first versions of CCSP were written for SPARC and Alpha, and where only ported to IA32 based architectures much later. In the meantime the IA32 architecture has become the only platform fully supported by CCSP as the ubiquity of IA32 based computers increased.

CCSP currently has good support for single-core as well as multi-core and SMP systems. It has a sophisticated scheduler [RSB09, Vel98] that dynamically creates batches of concurrent occam-pi processes which are split across operating system processes, enabling them to execute in parallel on multiple processors. The occam-pi processes are load-balanced dynamically by determining process locality based on frequency of communication between processes and by work "stealing".

CCSP has support for CIF [Bar05] — a novel way for interfacing occam-pi and C. CIF allows occam-style processes to be written in C and provides ways to communicate on channels with other C or occam processes. CIF on its own comes with the usual dangers of programming in C, and does not allow for the checking of any possible concurrency errors at compile time.

CCSP also offers very high levels of performance in terms of context switching, and execution as occam and CIF programs are compiled to the processors native assembly, and critical parts of CCSP are hand optimised for performance. CCSP is currently used for research in the modelling of complex systems (such as TUNA [tun05], CoSMoS [WSST08] and as the core of the RMoX [BJV03] operating system).

2.3.3 The Transterpreter

The Transterpreter, whose name is derived from "a Transputer Interpreter", is a virtual machine designed to run the extended transputer instruction set (ETC). The Transterpreter was designed to be a portable runtime for the occam-pi programming language. The choice of byte code was not difficult, as OCC21, the occam-pi compiler already targeted this virtual assembly language. ETC is well documented as it was based on the

Transputer instruction set which is described in detail in "Transputer instruction set: a compiler writer's guide" [INM88]. Changes and additions to the original instruction set are documented in the source of the IA32 translator for ETC in the form of C code as well as Barnes' and Jacobsen's theses [Bar03a, Jac08]. The runtime is written to include support for all of occam-pi, but is configurable to allow various features of the language to be disabled in the runtime, predominantly to save memory on small platforms, or to exclude them should they not be necessary for a given application.

For example, support for dynamic parts of the language can be disabled, removing the need for a memory allocator reverting the language to its static occam2.1 roots. It is also possible to disable support for barriers or the support for floating point instructions when floating point is not required. As an example, the first version of the Transterpreter that was built for the LEGO Mindstorms platform needed to execute in a footprint of less than 32KB memory. The virtual machine as well as user program and data needed to fit into this memory space. In this case, all extraneous features where removed to save on memory, thereby limiting certain features of the language and restricting it to features only available in occam2.1. A version of the Transterpreter also exists with a customised scheduler that allows multiple Transterpreters instances to schedule occam processes across multiple processors using POSIX threads to take advantage of SMP machines where all of occam-pi's features are desirable.

The Transterpreter is written in strict ANSI-C and consists of a core library which contains the interpreter and scheduler, and a platform dependent wrapper which needs to provide the abstraction for the Transterpreter to execute and interact with the platform it is running on. To date the Transterpreter has been ported to at least ten systems including Windows, OSX (PPC and x86), Sparc/Solaris, Linux (Sparc/Mips/x86/Arm), usually only requiring one to two weeks to port. Depending on the platform, additional time needs to be spent writing support libraries, such as in the case of the LEGO Mindstorms robotics kit, where interface code was required to control the robots motors and receive input from sensors. Other experimental ports have been prototyped including Symbian-based mobile phones and the Gameboy Advance both with the intention of using them as control platforms for robots. These were deferred for later completion in

favour of a more interesting platform that became available at the time, IBM's Cell BE processor.

The Transterpreter also provides the means of using C libraries from occam-pi through its foreign function interface (FFI) [Woo98]. The FFI can simplify the low-level control of hardware on embedded platforms, but more generally allows the language to make use of existing C libraries. The C interface and an automated tool for generating interfaces to C libraries is described in Chapter 3. The Transterpreter currently lacks support for CIF, however a CIF compatible interface for it is planned.

One of the more interesting aspects of the Transterpreter is that it is able to provide an execution environment for occam-pi programs that is itself written in occam-pi. A small "operating system" in the form of a network of occam-pi processes is compiled into the runtime and provides services to the user's program. For example, the POSIX compatible wrapper for the Transterpreter implements the runtime's deadlock detection, keyboard input and stdout/stderr functionality. This execution environment is adapted to target platforms as part of the porting process to provide any services relevant to a platform as part of the runtime. The idea of a rich operating environment is discussed in Chapters 3 and 4.

Newer versions of the Transterpreter have been modified to allow dynamic loading and unloading of occam-pi programs, as well as the execution of multiple occam-pi programs on the same virtual machine [RS08]. This is implemented using a richer occam-pi execution environment and is loaded instead of the usual "operating system". This has numerous applications, such as adding the ability to debug parallel programs by inspecting or logging the state of a running program and allowing real-time visualisation of program execution. Applications of this are discussed further in Chapter 4. The Transterpreter is primarily used in education and for research in robotics and visual programming [SJ08].

	C interface	CIF interface	Multiprocessor scheduling	Availability	Memory requirements
Trans-terpreter	Yes	No	Experimental	Most platforms	16kb
CCSP	Yes	Yes	Yes	IA32	70kb

Table 3: An overview of the occam-pi runtimes.

2.3.4 Summary of Runtimes

CCSP and the Transterpreter both provide reasonable choices for use as runtimes for occam-pi programs. The choice of runtime for a user will mostly be dictated by the application requirements. An overview of the runtimes is shown in Table 3.

2.4 Compilers

Three compilers are currently available for occam-pi, in various stages of completion and maturity. OCC21 and Tock [SB08b, SB08a] can be considered as stable, the third compiler NoCC, while interesting, is still too experimental to be considered further.

2.4.1 OCC21

OCC21 is based on the original occam2.1 compiler that was written for the Transputer. As such, the output that it produces is a superset of the assembly that Transputers used to execute, first described in [Poo98] as Extended Transputer Code or ETC. The TCE output from OCC21 can either be run through a specialised linker to be translated into a format that the Transterpreter can execute or translated into IA32 assembly using Tranx86 [Bar01].

When used in conjunction with the CCSP runtime Tranx86 is used to translate the ETC byte code to IA32 assembly. The combination of the components OCC21, CCSP and Tranx86, as well as the Transterpreter is commonly referred to as KRoC. KRoC also provides the necessary infrastructure for compiling CIF and occam-pi programs when used with CCSP.

KRoC also comes with an extensive compiler test suite, CGTests, which is used to ensure that OCC21 does not deviate from its original specification. The same test suite can also used to test the correctness of a runtime's implementation, such as the Transterpreter or CCSP.

OCC21 not only performs syntax checks, but also goes to great lengths to enforce parallel safety rules. Compile time verification includes banning shared access to variables and channels, ensuring that channels are used in the correct direction, verifying that a program is free of variable aliasing and providing warnings when variables are used with undefined values. These compile time checks ensure that a program is free of aliasing errors and race hazards, helping a programmer create correct programs by removing a range of common errors.

2.4.2 Tock

Tock is a new nanopass [SB08a] compiler written in Haskell, and is based on the ideas in the experimental 42 compiler that are described in 6.2. Like OCC21, Tock passes the all of the CGTests [INM10], the extensive occam-pi compiler test suite. occam-pi programs compiled by Tock are translated to the CIF syntax following a series of syntax and parallel safety checks, and then compiled using a C compiler and linked against the CCSP runtime.

Tock has advanced verification features built in that can detect a number of common concurrency errors at compile time, including all of the checks that OCC21 performs. In addition to OCC21's verification features, Tock is able to determine parallel usage correctness for dynamically sliced arrays, where the size of each array slice is determined at runtime. This is done through the use of the omega test [Pug92]. This is a compelling feature that makes Tock an interesting candidate for adding a Transterpreter compatible back-end to the compiler, to ensure a higher quality of code generation.

The advantages of using CIF as a back-end are performance and portability. In order to port the Tock tool chain, only CCSP needs to be ported to the new architecture, as

the C code that Tock generates is portable. Tock also has a strong performance advantage over OCC21 compiled programs since C compilers produce much more optimised assembly than Tranx86 for straight-line computation. Because no ETC byte code is produced, Tock currently has no support for the Transterpreter as a back-end, and as such is not used throughout this thesis, although it is considered in future work.

2.5 Motivating occam-pi usage today

occam-pi is a language designed for running on multiprocessor systems, and has strict rules about concurrency built into the language. These rules allow for compile time checking of common concurrency problems. The language can also be formally reasoned about using CSP.

The Cell BE processor, in particular the SPU processors bears a resemblance to the Transputer — they have isolated local memory and form a Multiple Instruction, Multiple Data (MIMD) "network" of processors interconnected by high speed links. SPUs are bootstrapped by loading programs onto them using a host processor (the PPC) and the host processor is used for handling any operating system related tasks. While there are differences in architecture, such as main memory being accessible to all of the Cell's processors — to the host processor as part of its memory map and to the SPU via through DMA controllers — some of the Cell's interconnect features make it possible, and potentially desirable, to bring occam-pi to it.

The Transputer provided hardware support for many of the services occam requires, which have now been replaced by software in the form of CCSP and the Transterpreter, such as scheduling and interrupts on inter process communication. Adding these features to SPUs in software in the form of a micro-kernel like the Transterpreter would allow the Cell BE to function like a network of Transputers.

The advantages of such an approach are:

- Programs written in occam-pi have the benefit of compile-time checking for concurrency, making it potentially safer to program for the Cell.

- Work and communication can be overlapped on an SPU. By having concurrent processes executing on each SPU computation and communication can be executed in parallel making full use of the bandwidth of the Cell.

- Programs written for the Cell should be portable to other systems and vice versa.

Programming modern multi-core systems like the Cell is an ongoing concern. Announcements of current and future processors by the world's largest vendors indicate that processors are developing towards "a multiple-small cores with local memory" model similar to the Cell, such as in today's graphics processing units.

Current limitations in memory bandwidth is the primary stumbling block to high performance computing. The Cell's architecture is the first step in solving this problem. The need for providing software developers with alternatives to today's programming languages based on single-threaded paradigms is clear. This thesis explores such a language, along with the environment surrounding the language which will make it a compelling tool for future software development.

There are three key aspects which need to be addressed in the occam-pi language in order to make it a better general purpose programming language for modern computer systems, particularly for the Cell. First the language has to have a reliable and reusable means for interfacing with existing software and libraries, as presented in Chapter 3. Second better support for the underlying hardware needs to be provided both in the language (presented in Chapter 5) and third the Transterpreter runtime (presented in Chapter 4). There is a strong overlap between these aspects. Libraries are often considered as part of a language itself. A language's functionality also depends on services provided by the runtime that cannot come in the form of a library. These three aspects are discussed over the course of this thesis.

Chapter 3

occam-pi as a control language

This chapter presents a tool for rapid and automated wrapping of C libraries for occam-pi. It provides both a valuable tool for the end user as well as important support for the work presented in this thesis. The tool has also been used in new occam-pi research and development, which is detailed further in Section 3.10.1.

The tool presented in this Chapter builds on the existing foreign function interface (FFI) [Woo98] that exists in occam-pi. By automating the wrapping of foreign libraries, access can be provided to a large existing code-base from within occam-pi without a prohibitively large investment in time. Both language developers and users will benefit as both will be able to add support for new libraries easily.

SWIG (Simple Wrapper and Interface Generator) [Bea96] is a multi-language foreign function interface generator, providing the infrastructure needed for automatic library wrapping. Support for the generation of wrappers for individual languages is provided by self-contained modules, which are added to SWIG. The remainder of this chapter deals specifically with the occam-pi SWIG module[1] created as part of this thesis work.

[1]The occam-pi module is not at the time of writing part of the official SWIG distribution, it is however available from http://projects.cs.kent.ac.uk/projects/swigoccam/trac/ as source.

3.1 Background and motivation

It is not always feasible to program an entire solution to a problem in occam-pi. Applications needing to deal with file I/O, graphical user interfaces, databases or operating system services, are not possible unless a very large existing code-base is rewritten in occam-pi or made available to occam-pi through existing foreign libraries implementing such services.

Therefore occam-pi's foreign function interface exists to enable programmers to reuse existing C code or write certain portions of an application in C. It does, however, require a large amount of wrapper code to interface a library with occam-pi. This may not be a big problem when dealing with small amounts of code, but writing a wrapper for even a relatively modest library can quickly become time-consuming and tedious, and therefore, error-prone. This becomes a further problem when one considers that the library being wrapped can evolve over time, and the wrappers must be updated to reflect changes in the library in order to be useful.

As an example two GUI frameworks using hand-wrapped C bindings for occam-pi where started and brought to proof-of-concept stages without automated tools to assist in the wrapping process. Sadly these framework where never completed. Both projects indicated the excessive time and effort required to provide a complete library wrapping as reasons for only implementing a subset of the libraries functionality. The first library was GTK [GTK05], a popular open source GUI toolkit, which was used as the basis of an occam2.1 library wrapper and drawing manager [Whi00]. The other was for the X11 Window Gadgets toolkit [Bar03b, Sto03] that, in addition to generating occam-pi wrappers, implements an advanced rendering manager using occam-pi's mobility features. While more effort was put into creating powerful programming models for the graphics systems than creating wrappers, the time taken to create interfaces to the subset of functionality that was made available could have been better spent developing the frameworks further.

Without better access to existing libraries and code, it may be difficult to argue

that occam-pi is a better choice for developing large, complex systems than other languages. In order to leverage the large amount of existing work and infrastructure that is provided through operating system and other libraries interfacing with C must be simplified. It is imperative for the success of occam-pi that it is able to make use of the large amount of existing work, in the form of system, graphical, database, and highly optimised scientific libraries which are readily available, and which will add important functionality to occam-pi.

occam-pi is a language suited to the parallelisation of existing programs. A number of other high level languages (such as Python or Java) are used as control or infrastructure languages for legacy C code, allowing the user to combine the ability to express concepts more concisely than would be possible in C, while still harnessing the speed and power of the existing C code. The existing C code may be in the form of libraries, legacy applications, or new code specifically written for an application to speed up key parts. High level languages are able to provide features not found in C such as pattern matching and higher order data structures. In addition occam-pi provides a powerful set of concurrency primitives that are not available in C and hence can be used to solve problems that would otherwise be difficult to express in C.

Software written in higher level languages is often considered to be simpler to maintain than lower level programming languages in terms of the infrastructure that one is able to create with them. Beazley [Bea97] stated that the overall quality of software, including that of the faster, lower level code was improved through the use of a stricter, more structured control language, Python, as the legacy code is adapted to work better with the control infrastructure. The intent is to achieve a similar result with occam-pi.

3.2 Integrating libraries into occam-pi

C libraries are usually written for sequential, event driven programs and often lack support for parallel paradigms. When used from occam-pi it is desirable to have an occam-pi-like framework built around them which is more in line with a process-oriented programming model.

In contrast to the more commonly used event model, which languages such as Java or C use, the process oriented programming model arguably encourages a more realistic description of real world events by allowing processes to react to specific events on distinct event channels, as opposed to having a single event-queue for all events.

The commonly used event model tends to use potentially hard to track callback mechanisms for processing events. For example, Java's SWING has a single very large event loop which, on receiving an event, triggers a call-back and runs a function or method assigned to deal with that event. Programs written using this type of single-event driven model end up being structured in a manner that only allows one event to be processed at a time. A program's flow is then traceable back to a single event and runs into problems if more events occur while the previous event is still being processed. The single-event model is arguably an oversimplified representation of the real world, in which multiple events can occur simultaneously. A single event loop is unable to deal with simultaneous events and often lacks the means to assign priority to events should there not be enough resources to process them simultaneously. In terms of concurrency, these are pitfalls that can lead to programs that are difficult to write and maintain and which can suffer from race conditions and deadlock. Arguments have also been made suggesting that concurrency should not be implemented in the form of libraries [Boe05], giving further support for languages like occam-pi which have concurrency primitives.

occam-pi's equivalent "event model" is implemented using occam-pi's channel communication which permits mapping, as well as prioritising of specific events. It is possible to set up a system which can either have separate processes (independent 'event loops') always ready to intercept events or to have a system which is able to prioritise on multiple simultaneous events. This type of a system is arguably better suited to describing and processing events in the real world than the sequential event models implemented by languages such as Java or C as it can mimic concurrency in the real world.

occam-pi's compiler and concurrency primitives provide a degree of security and error checking that C or Java cannot. Being able to make use of existing C code from

a programming language that is able to provide such guarantees is likely to reduce the scope for common concurrency errors.

When creating occam-pi interfaces to C libraries which use an event loop or call-back based mechanism it is possible to abstract over the common event model by having separate processes or channels which respond to specific events. As well as enabling wrapping of C libraries for occam-pi it would be beneficial to integrate the wrapped libraries to fit better with the occam-pi programming model. Being able to abstract control logic into separate independent components can allow a programmer to create software in a more structured manner. Such abstraction can be built faster with the help of automated tools by generating an interface to existing libraries, which can then be used as the foundation for a process oriented framework for using these libraries.

3.3 Interface Generators

The work presented is not the first attempt to provide automatic wrapping of foreign library code for occam-pi. The occam to C interface generator Ocinf [Lew96] was the first widely available interface generator for occam. Ocinf generates the glue code to wrap C functions, data structures and macros so that they can be used from occam2.1. Since the occam-pi syntax is a superset of occam2.1 and they share the same FFI mechanisms, it would still be possible to use Ocinf to generate interfaces for occam-pi. Ocinf, however, has not been maintained since 1996 and relies on Lex and Yacc [JL92]. It has proven difficult to get Ocinf to work, since Lex and Yacc have evolved and no longer support much of the old syntax. Making the Ocinf code base work with the current versions of Lex and Yacc would require rewriting significant portions of 7,000 lines of Lex and Yacc source. With the emergence of SWIG as the de facto standard in open source interface and wrapper generation, upgrading the Ocinf tool to work again did not present itself as a viable option.

The SWIG framework is a general purpose code generator which has been in constant development since 1996. It currently allows more than 11 languages to interface with C and C++ libraries, including Java, Perl, Python and Tcl. SWIG is a modular

framework written in C++ that by implementing a new language specific module provides the means for adding interface generation support for virtually any programming language with a C interface. Additionally, SWIG comes with good documentation and an extensive test suite which can help to ensure higher levels of reliability.

Uses of SWIG range from scientific computation using Python [Bea97] to an online-business such as Google [Ste05]. While other interface generators exist, they are generally language-specific and not designed to provide wrapper generation capabilities for other languages. SWIG was from the outset designed to be language-independent, and this gives it a wide and active user base.

3.4 SWIG Module Internals

SWIG itself consists of two main parts; an advanced C/C++ parser, and language specific modules. The C/C++ parser reads header files or specialised SWIG interface files and generates a parse tree. The specialised interface files can provide the parser with additional SWIG specific directives which allow the interface file author to rename, ignore or apply contracts to functions, use target language specific features or otherwise customise the generated wrapper.

The language specific module inherits a number of generic functions for dealing with specific syntax. Functions are overloaded by the modules and customised to generate wrapper code for a given language. The actual wrapper code is generated after the parse tree has undergone a series of transformations. Each transformation may be handled by SWIG's core or may be overridden by a language specific module. Library functions are provided to allow easy manipulation and inspection of the parse tree. The parse tree consists of a set of nodes which are essentially hash-tables. The SWIG documentation [Bea05] provides more detail about how SWIG functions and how one would go about writing a new language module.

3.5 An Overview of the occam-pi FFI

The occam-pi FFI requires that foreign C functions be wrapped up in code that informs the occam-pi compiler OCC21 [WMBW00] how to interface with a given foreign function. This is required as the calling conventions for occam-pi and the C code differ. The wrapper essentially presents the arguments on the occam-pi stack to the external C code in a manner that it can use. The wrapping process is illustrated with the following C function:

```
int aCfunction(char *a, int b);
```

Listing 3.1: A C function prototype

occam-pi performs all FFI calls as a call to a function with only one argument, using C calling conventions. The argument is a pointer into the occam-pi stack, in which actual arguments reside, placed there by virtue of the external function signature provided to the occam-pi compiler. The arguments on the stack are all one word in length, and the pointer into the stack can therefore conveniently be accessed as an array of **int**s. In order to correctly access an argument, it must first be cast to the correct data-type, and possibly also dereferenced in cases where the argument on the stack is a pass by reference argument — a pointer to the actual data.

```
void _aCfunction(int w[]) {
  (int *)(w[0]) = aCfunction((char *)w[1], (int)w[2]);
}
```

Listing 3.2: Converting occam-pi pointers for use in C

The code 3.2 defines an occam-pi callable external C function with the name `_aCfunction` which takes an array of **int**s and always has a void return type. This function's job is to call the real `aCfunction` with the provided arguments, which then performs the actual work.

The array passed to `_aCfunction` contains pointers to data or in some cases the data itself, which is to be passed to the wrapped function, as well as a pointer used to capture the return value of the called function. While a function in C may have a return

value, external functions in occam-pi are presented as **PROC**s which may not return a value directly. Instead references (addresses) of variables can be used for this task. In cases where a function has no return value one simply omits the use of a reference variable to hold the result of the called external function.

In essence, the wrapper function just expands the array **int** `w[]` to its sub components and type casts them to the correct C types that the wrapped function expects. The occam-pi components that completes the wrapping are defined as follows:

```
#PRAGMA EXTERNAL "PROC C.aCfunction(RESULT INT result,
                                    BYTE a, VAL INT b) = 0"

INLINE PROC aCfunction(RESULT INT result, BYTE a, VAL INT b)
  C.aCfunction(result, a, size)
:
```

Listing 3.3: Calling C from occam-pi

The first component is a **#PRAGMA** compiler directive which informs the compiler of the name of the foreign function, its type and parameters. The **#PRAGMA EXTERNAL** directive is similar to C's **extern** keyword. Function names are prefixed with one of "C.", "B.", or "BX." for a standard blocking[2] C call, a non-blocking C call and an interruptible non-blocking C call respectively. This prefix is used to determine the type of foreign function call, and is not used when determining the name of the external C function, which should in fact be prefixed with an underscore instead (regardless of its type): the **PROC** `C.aCfunction` will call the C function called `_aCfunction` as will the "B." and "BX." **PROC**'s.

The second **PROC** is optional and serves only to provide a more convenient name to the end user by presenting the wrapped function without the call type prefix. While this is not strictly necessary, it enables the wrapper to provide an interface which follows the wrapped library closer.

[2]by blocking, we mean blocking of the occam-pi runtime kernel.

From the example above it should be clear that manually producing wrapper code for a small number of functions is not a problem. However writing such code for larger bodies of functions is laborious and error prone. The OpenGL [MWS99] library is prime example of a library where automation is preferred, as the library consists of over five hundred functions and is constantly growing.

More information on how to use KRoC's foreign function interface and various types of system calls can be found in Wood [Woo98]. Details of performing non-blocking[3] system calls from KRoC can be found in Barnes' [Bar00].

3.6 SWIG and the occam-pi FFI

3.6.1 Using SWIG to Generate Wrappers

The wrapper code generated by SWIG is much the same as one would generate by hand as demonstrated above. This section provides more detail on how the occam-pi SWIG module performs the mapping from the interface file to the generated wrapper.

3.6.2 Generating Valid occam-pi PROC Names

In order to allow C names to be mapped to occam-pi, all '_' characters must be replaced by '.' characters. This is done as the occam-pi syntax allows '.' but not '_' characters in identifier names. A function like `int this_func(char a_variable)` would map to `PROC this.func(RESULT INT r, BYTE a.variable)`. The only real effect this has is on function and `struct` names since parameter names are not actually used by the programmer.

3.6.3 Auto-generating Missing Parameter Names

SWIG needs to generate parameter names automatically for the occam-pi wrappers should they be absent in function definitions. Consider a function prototype such as:

[3] they execute in a separate os-thread, not blocking the occam runtime.

```
int somefn(int, int);
```

SWIG will automatically generate the missing parameter names for the **PROC**s which wrap such functions. This does not affect the user of the wrappers, as the parameter names are of no importance, other than possibly providing semantic information about their use. Parameter names are however necessary in order to make the occam-pi wrapper code compile.

The code listed in above would map to a **PROC** header similar to:

```
PROC somefn(RESULT INT ret.value,
    VAL INT a.name0, VAL INT a.name1)
```

The occam-pi module for SWIG generates unique variable names for all auto-generated parameter names in **PROC** headers, ensuring that there is no parameter name collisions.

3.6.4 Data Type Mappings

The mapping of primitive C data types to occam-pi are straightforward, as there is a direct association from one to the other. The mappings are based on the way parameters are presented on the occam-pi stack during a foreign function call. For example an occam-pi **INT** maps to a C **int**$*$ (that is, the value on the occam-pi stack is a pointer to the pass-by-reference **INT** and dereferencing is needed to get to the actual value). The complete set of type mappings can be found in [Woo98].

3.6.5 Structures

C's **struct**s can be mapped to occam-pi's **PACKED RECORD**s. Ordinary occam-pi **RECORD**s cannot be used, as the occam-pi compiler is free to lay out the fields in this type of record as it sees fit. **PACKED RECORD**s on the other hand are laid out exactly as they are specified in the source code, leaving it up to the programmer (or in this case SWIG) to add padding where necessary. As an example, the following C **struct**:

```
struct example {
  char  a;
  short b;
};
```

would map to the following **PACKED RECORD** on a 32 bit machine:

```
DATA TYPE example
  PACKED RECORD
    BYTE  a:
    BYTE  padding:
    INT16 b:
:
```

This handling of **struct**s is somewhat volatile as it makes assumptions about the layout of C **struct**s and generally assumes the layout generated by current GCC compilers. A **struct**s memory layout may change depending on the compiler or architecture and may also be dependent on word-size and endianess. This makes the use of **struct**s a potential hazard when it comes to the portability of the generated wrapper. Currently padding is only supported for 32 bit architectures. Support for architectures with different word sizes can be added in the future, however as this is dependent on matching the memory layout of a structure that could change in future compilers, this option should be used with caution, and only in cases where the added performance is absolutely necessary.

While the above approach permits very fast (direct) access to data in **struct**s, in order to ensure portability it is better to use to use the set of C accessors and mutator functions automatically generated by SWIG for structures. These can be used by the occam-pi program to access and mutate a given structure. The command line parameter "-structpointers" generates *get* and *set* functions for all members of a structure and stores pointers to a structure in an occam-pi **INT**. For example, the get **PROC** for the member `a` in the structure `example` would be:

```
DATA TYPE example IS INT:
PROC get.example.a (RESULT BYTE result, example name)
```

where `example name` is the pointer to a structure of type example and `result` is where the value of the member 'a' will be stored.

It should even be possible for SWIG to produce code to automatically convert from an occam-pi version of a structure to a C version (and vice versa), in order to provide more transparent **`struct`** access to the end user. This would of course be significantly slower than mapping the structures directly into occam-pi **`PACKED RECORD`**s.

3.6.6 Unions

A C **`union`** allows several variables of different data types to be declared to occupy the same storage location. Since occam-pi does not have a corresponding data type, a workaround needs to be implemented. The occam-pi **`RETYPES`** keyword allows the programmer to assign data of one type into a data structure of another. As an example, a **`struct`** that is a member of a **`union`** could thus be retyped to an array of bytes. This is useful since the **`PROC`** wrapping a function that takes a **`union`** as one of its arguments can take an array of **`BYTE`**s which are then cast to the correct type in the C part of the wrapper. This means that any data structure which is a member of a **`union`** can be easily passed to C. Functions which return **`union`**s can return a **`char`** $*$ which can then be retyped to the corresponding structure in occam-pi. The remaining difficulty with this approach is that occam-pi programmers need to make sure that they are retyping to and from the correct type of data, as it is easy to assign the **`BYTE`** array mistakenly to the wrong data structure and vice versa. The occam-pi compiler, like the C compiler, is unable to check for the correctness of such an assignment.

The SWIG occam-pi module generates *create* and *delete* wrapper functions for unions, which return a pointer to the union. This can then be used with functions that take such a union as a parameter. Currently it is not possible to manipulate a union directly, without casting it into a different data type, such as a struct, and then using the structs accessor methods to manipulate it.

3.6.7 Pointers and Arrays

In C, an array can be specified using square brackets, `int a[]`, or via a pointer, `int *a`. It is not always possible to know if a pointer is just a pointer to a single value, or in fact a pointer to an array. Different wrapping functions are needed in each case.

By default pointers are treated as if they are not arrays, and mapped into a pass by reference occam-pi parameter. If a parameter actually does refer to an array, it is possible to force SWIG to generate a correct wrapper for that function by pre-pending 'array_' to the parameter name in the interface file. Examples of this are provided throughout the chapter where appropriate. When the array is declared using the square brackets syntax, the occam-pi module automatically detects that the parameter is an array and no annotation is required. Similarly multi-dimensional arrays specified with square brackets are dealt with correctly as well as arrays with fixed size values, such as `int a[3]`.

3.6.8 Typeless pointers

The current default behaviour for type mapping `void *` to occam-pi is to use an `INT` data type. Since `void *` can be used in a function which takes an arbitrary data type, this restricts the use of `void *` somewhat to only allowing `INT` to be passed to a function. The following example demonstrates the mapping of a typeless pointer to an array in the OpenGL `glCallLists` function:

```
void glCallLists(GLsizei n,
                 GLenum type,
                 const GLvoid *lists);
```

Listing 3.4: glCallLists original prototype.

```
void glCallLists(GLsizei n,
                 GLenum type,
                 const GLvoid *array_lists);
```

Listing 3.5: glCallLists prototype modified for SWIG-occam-pi.

In Listing 3.5 the 'array_' string is prepended to the 'lists' variable name, to indicate that it receives an array, in contrast to Listing 3.4 which would result in the wrapped function expecting a single value. The `GLsizei n`[4] variable tells the function the size of the data type being passed to it, the `GLenum type` variable specifies the type being passed into the `GLvoid *array_lists`, so that the function knows how to cast it and use it correctly. Since the default type mapping behaviour here is to type map a C **`void *`** to an occam-pi **`INT`**, or `[]`**`INT`** if it is an array, the ability to pass it data of an arbitrary type is lost. So, when calling `glCallLists` from occam-pi one always has to specify that one is passing an integer to `glCallLists`, by passing the correct **`enum`** value. Here is an example of calling `glCallLists` from occam-pi:

```
PROC printStringi(VAL []BYTE s, VAL INT fontOffset,
                  VAL INT length)
  MOBILE []INT tmp:
  SEQ
    tmp := MOBILE [length]INT
    SEQ i = 0 FOR length
      tmp[i] := (INT (s[i]))
    glPushAttrib(GL.LIST.BIT)
    glListBase(fontOffset)
    glCallLists(SIZE s, GL.UNSIGNED.INT, tmp)
    glPopAttrib()
:
```

Listing 3.6: An example of `glCallLists` used to render text in occam-pi.

It is possible, by manually writing some C helper functions, to allow the end users of a library to pass a greater range of data types to functions taking **`void *`** parameters. This can be done by writing proxy **`PROC`**s for every type of data that the original function accepts. Each proxy **`PROC`** then calls one of the auto-generated **`PROC`**s that will always expect `[]`**`INT`** data, with the appropriate parameters. In this way it would, for example,

[4] `GLsizei`, `GLenum` and `GLvoid` are simply C **`typedef`** declarations, mapping them to **`int`**, **`enum`** and **`void`** types respectively. These are used to enable more architecture independent code, should types work slightly differently on other platforms.

be possible to provide a **PROC**ess which accepts REAL32 data, as shown in Listing 3.7, but converts it to the type that the automatically generated wrapper function requires by using the **RETYPES** syntax.

Access to **PROC**s accepting other types can be provided in a similar manner. It may even be possible to let SWIG automate much of this work by using its macro system, although it is likely this would need to be customized on a per-library basis.

```
PROC glCallLists.R32(VAL INT n, []REAL32 lst):
  []INT intlist RETYPES lst:
  glCallLists(n, GL.FLOAT, intlist)
```

Listing 3.7: Data-type aware wrappers for functions that take **void** parameters.

3.6.9 Enumerations

C **enum**s allow a user to define a list of keywords which correspond to a growing integer value. These are wrapped as series of integer constants. So for an **enum** defined as:

```
enum myenum {
  ITEM1 = 1,
  ITEM4 = 4,
  ITEM5,
  LASTITEM = 10
};
```

Listing 3.8: An example C **enum**.

the following occam-pi code is generated:

```
-- Enum myenum wrapping Start
VAL INT ITEM1 IS 1:
VAL INT ITEM4 IS 4:
VAL INT ITEM5 IS 5:
VAL INT LASTITEM IS 10:
-- Enum myenum wrapping End
```

Listing 3.9: Code generated by SWIG for an **enum**.

If several enumerations define the same named constant, a name clash occurs when the wrapper is generated. If this is a problem, it is possible to change the names of the **enum** members in the interface file (while not affecting the original definition). This will in turn affect the names of the generated constants in the wrapper, thus making it possible to avoid name clashes.

Should **enum** name clashes be a regular occurrence, it would be possible to implement an option of naming the **enum**s differently to ensure that the wrapped constants have unique names. For example, the wrapped constant names on the preceding page could be generated using the **enum**'s name as a prefix to the constants name. The programmer using the wrapped code would have to be aware of the convention used in the names of **enum** constants. This, second option is now implemented in the latest version of the SWIG occam-pi wrapper.

3.6.10 Preprocessor directives

C's **#define** preprocessor directives are treated similarly to **enum**s. Any value definitions are mapped to corresponding constants in occam-pi. More complex macros are currently ignored, although there may be a way to wrap them correctly using occam-pi preprocessor directives. Listings 3.10 and 3.11 show how some **#define** statements map to occam-pi. Currently only preprocessor directives that declare constants are mapped to occam-pi as it may not make sense to map more complex directives to occam-pi as C macros permit the developer to do many transformations that are not permitted in occam-pi.

```
#define AN_INTEGER 42
#define NOT_AN_INTEGER 5.43
```

Listing 3.10: Sample preprocessor directives in C.

```
VAL INT AN.INTEGER  IS 42:
VAL REAL64 NOT.AN.INTEGER  IS 5.43:
```

Listing 3.11: Preprocessor directives translated to occam-pi.

3.6.11 Wrapping Global Variables

SWIG's default behaviour for wrapping global variables is to generate wrapper functions, which allow the client language to get and set their values. While this is not an ideal solution in a concurrent language, as one could be setting the global variable in parallel leading to race conditions and unpredictable behaviour, it is the simplest solution. occam-pi itself does not normally allow global shared data other than constants.

There are plans to address this issue by adding functionality to the SWIG occam-pi module, which will allow the usage of a locking mechanism, such as a semaphore, to make sure that global data in the C world does not get accessed in parallel. The wrapper generator could generate two wrapper **PROC**'s for getting and setting the global variable, as well as a third **PROC**, which would need to be called by the user to initialise the semaphores at start-up. occam-pi provides an easy to use, lightweight semaphore library, and it would therefore be easy to manage access to global data from occam-pi.

If a library is not itself thread safe, the end user of the library currently needs to be aware of the dangers presented by global shared data, and use an appropriate locking mechanism. There is currently no way of determining such danger automatically.

3.6.12 Non-blocking C calls

Non-blocking foreign function calls can be automatically generated by calling SWIG with the command-line parameter "-gen_blocking", which generates three **PROC**'s prefixed with "C.", "B.", or "BX." respectively for each function, providing all wrapped functions with a non-blocking interface. Using only this command-line option provides the user with the usual prefixless **PROC** wrapper which calls the "C." prefixed **PROC** by default. In order to provide a clean interface to functions which should be non-blocking, the SWIG interface file can be annotated to specify which type of call (blocking/non-blocking/altable-blocking) should be used in the prefixless, proxy PROC that is generated. For a function which needs to be non-blocking one can annotate the interface file using SWIG's `%feature` directive:

```
%feature("blocking") itakelong(int) "B"
void itakelong(int a);
```

The `%feature` directive annotates the parse tree for the function `itakelong` with the key "blocking" which has the value "B". The SWIG occam-pi module checks for the blocking key and when found, generates the following occam-pi code:

```
#PRAGMA EXTERNAL "PROC B.itakelong (VAL INT a) = 3"

INLINE PROC itakelong (VAL INT a)
  B.itakelong (a)
:
```

Specifying the value "BX" instead of "B" as the value for the "blocking" `%feature` directive will create a "BX" prefixed **PROC**. Using the `%feature` directive to specify if a function should be non-blocking is independent of, but can be used in combination with, the "-gen_blocking" command-line parameter. It should be noted that at the time of writing neither of the two blocking functions are implemented in the Transterpreter runtime and should not be considered supported there.

3.6.13 Wrapper Documentation

The SWIG module for occam-pi also automatically generates OccamDoc [occ09] documentation for all of the code that it generates. This can be parsed by the OccamDoc program and made available on-line in the form of web pages.

3.7 Using SWIG

By running a command on the command line you can generate wrappers from a C header file that describes the functions and data-structures exposed by a library. This is done by running the command:

```
$ swig –occampi –module myheader myheader.h
```

where `myheader.h` is the C header file that contains the function definitions that you would like to make use of from occam-pi.

In many cases it is enough to simply point SWIG at a C header file and have it generate a wrapper for occam-pi from that header file. However it is generally better to take a copy of the C header (.h file), and copy that to create a SWIG interface (.i) file. For example, when wrapping the OpenGL library, `gl.h`, the OpenGL header file, would be copied to `gl.i`. SWIG specific directives can then be added at the head of the file, such as defining the name of the module, which will determine the names of the generated wrappers. Continuing with OpenGL as an example, we might add the following code at the top of the interface file:

```
%module gl
%{
#include <GL/gl.h>
%}
```

This names the SWIG generated wrappers "gl" and tells SWIG to include the code surrounded by parentheses **`#include`** `<GL/gl.h>` in the generated wrappers, so that, when compiled they are able to reference the targets original header files. To have SWIG generate a wrapper for occam-pi from the newly created .i, interface file the following command is run:

```
$ swig –occampi gl.i
```

Note, that the "-module" command line option shown in the first example is no longer needed, since the module name is specified by the `%module gl` in the interface file.

As an example of what might be changed in an interface file, the wrapping of OpenGL's `glCallLists` function is illustrated below. As the previous section already mentioned, the occam-pi module's default behaviour is to typemap pointers to **`INT`**'s in occam-pi and in order to correctly wrap pointers to arrays, an interface file must specify which pointers are pointers to arrays. This can be done by modifying the name of the variable that is to be typemapped by prefixing it with 'array_'. The `glCallLists` function call takes an **`int`**, an **`enum`** and an array of **`void`** $*$. To generate a correct

wrapper for the function one would modify the function declaration from:

```
void glCallLists(GLsizei n,
                 GLenum type,
                 const GLvoid *lists);
```

to the following:

```
void glCallLists(GLsizei n,
                 GLenum type,
                 const GLvoid *array_lists);
```

Once appropriate modifications are made to the .i files, SWIG is used to create the appropriate wrappers. When SWIG is run, it generates three files: `module_wrap.c`, `module_wrap.h` and `occ_module.inc`, where "module" is the name that the interface file specifies with the `%module` directive. The generated C files can then be compiled and linked into a shared library. On Linux one would run the commands:

```
$ gcc -I. -g -Wall -c -o module_wrap.o module_wrap.c
$ ld -r -o liboccmodule.so module_wrap.o
```

To use the generated wrapper from occam the .inc file needs to be included in the occam-pi program with the following directive:

```
#INCLUDE "occ_module.inc"
```

The newly created shared library needs to be linked with the KRoC binary as shown in the command below. This command may need to be modified to include the correct library include and linking path.

```
$ kroc myprogram.occ -I. -L. -loccmodule -lwrappedlibrary
```

SWIG has many other features which are not specific to the occam-pi module, designed to aid the interface builder in creating more advanced interfaces between higher-level languages and C. These are fully documented in the SWIG documentation [Bea05].

Updating an existing wrapper (.i) file is generally done by recreating the .i file from scratch. Any functions written to enhance the wrapper file can be kept in a separate .i file that includes the modified copy of the header file.

3.8 Examples of automatic library wrapping

This section will introduce two examples which demonstrate the relative ease with which SWIG can be used to wrap C library code to make it available to occam-pi. The first example is a fictitious maths library, followed by the real world example of the OpenGL library which shows that these concepts scale to large libraries.

3.8.1 A Simple Math Library Demo

This example uses a fictitious floating point library called "calc" which contains a range of standard floating point arithmetic functions. The following shows the header file "calc.h" for this library. For the purpose of wrapping the library, the contents of the "calc.c" file are of no importance other than in generating an object file.

```
float add(float a, float b);
float subtract(float a, float b);
float multiply(float a, float b);
float divide(float a, float b);
float square(float a);
```

Listing 3.12: Example C header file calc.h

To begin the wrapping process an interface file for SWIG is created from the calc.h header file. In order to create the interface file, the calc.h header file is copied to a new file named calc.i. This file is then modified to look like the Listing 3.13. The names of the generated wrapper files are indicated by the `%module` line in the interface file. The next three lines inform SWIG that it should embed the **`#include`** `"calc.h"` statement into the generated C header file. It is in the interface file that any additional information, such as whether pointers to data are arrays or single values must be included. In this case the only modifications to the interface file that are needed are the four lines of code added at the start of the file.

The C and occam-pi portions of the wrapper are generated by calling SWIG on the command line:

```
$ swig -occampi calc.i
```

```
%module calc
%{
#include "calc.h"
%}
float add(float a, float b);
float subtract(float a, float b);
float multiply(float a, float b);
float divide(float a, float b);
float square(float a);
```

Listing 3.13: Example SWIG interface file `calc.i`.

The occam-pi program "main" in Listing 3.14 demonstrates the use of the C functions, the file `occ_calc.inc` included at the start of the listing is the interface file generated by SWIG.

```
#USE "course.lib"
#INCLUDE "occ_calc.inc"
PROC main (CHAN BYTE kyb, scr, err)
  INITIAL REAL32 a IS 4.25:
  INITIAL REAL32 b IS 42.01:
  REAL32 result:
  SEQ
    add(result, a, b)
    out.string("Result of addition: ", 0, scr)
    out.real32(result, 0, 3, scr)
    square(result, a)
    out.string("*nResult of squaring: ", 0, scr)
    out.real32(result, 0, 3, scr)
    out.string("*n*n",0,scr)
:
```

Listing 3.14: Example program using SWIG generated code.

It is often convenient to crate a build script such as the one shown in Listing 3.15.

```
#!/bin/bash
gcc -c -o calc.o calc.c
gcc -c -o calc_wrap.o calc_wrap.c
ld -r -o libcalc.so calc.o calc_wrap.o
kroc calculate.occ -L. -lcalc -lcourse
```

Listing 3.15: A build script for swig-occam-pi.

That is all that is required for a simple set of functions to be wrapped.

3.8.2 Wrapping OpenGL

The development of the SWIG occam-pi module is motivated by a need for the broader availability of scientific, and system libraries on platforms such as the Cell. It is also motivated by the need for a robust graphics library for occam-pi in order to provide visual feedback which coincides with the overreaching goal of making occam-pi better suited to modern hardware. The OpenGL library is a good choice for wrapping since it is based on an industry-standard which is supported on most modern platforms, often with hardware acceleration. The OpenGL standard itself contains no window management functionality or support for GUI events and therefore requires the support of a different library to provide the functionality needed to open a window, establish an OpenGL rendering context and handling input. For window management, the SDL [Lan05] graphics and user interface library is used, due to its simplicity and high level of cross-platform compatibility. While GLUT [Kil96] — The OpenGL Utility Toolkit — is considered as an alternative, its call-back based model is more difficult to interface with than SDL's polling model. SDL also provides a wider range of features such as window management, input, sound and networking functionality. A subset of the SDL library is wrapped to allow the user to create and control windows as well as creating a rendering context for OpenGL.

In order to create the wrapper for OpenGL the header files `gl.h` and `glu.h` were copied to `gl.i` and `glu.i` respectively. The newly created .i files have the code shown in Listing 3.8.2 added to their headers (for `gl.i` and `glu.i`).

```
%module gl
%{
#include <GL/gl.h>
%}
...
```

```
%module glu
%{
#include <GL/glu.h>
%}
...
```

Table 4: GL/gl.h and GL/glu.h interface files for occam-pi

Subsequently a third file called `opengl.i`, shown in Listing 3.16, was created which linked the two previously created modules into one.

```
%module opengl
%include gl.i
%include glu.i
```

Listing 3.16: OpenGL interface file for SWIG.

Finally, the SWIG occam-pi was invoked on the command line to generate the OpenGL wrappers:

```
$ swig -occampi opengl.i
```

This generated three files, `opengl_wrap.c`, `opengl_wrap.h` and `occ_opengl.inc`. To make the OpenGL library accessible to an occam-pi program, the generated C wrapper files were compiled into a shared library and the `occ_opengl.inc` file was **#INCLUDE**ed in a program. A copy of the generated wrappers along with more detailed instructions on how to recreate them, as well as the SWIG-occam-pi module source code can be found at [Dim05].

A demostration of the OpenGL library wrappers can be seen in Sampson's occam-pi Game of Life implementation as described in [SWB05]. Figure 3 is an image of the running application, depicting a cellular automaton written in occam-pi. While some customisation is required in order to get the most out of a wrapper, it is still much less effort than creating such a wrapper by hand.

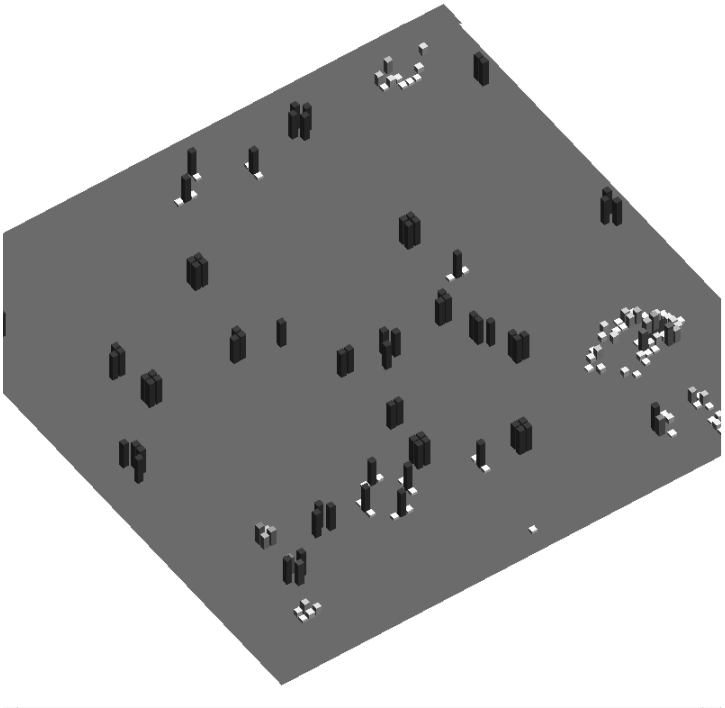

Figure 3: Lazy simulation of Conway's Game of Life.

3.9 Possible pitfalls when using C with occam-pi

A number of issues arise when interfacing with C functions from occam-pi. C, being a sequential language, lacks the built-in support for concurrency of occam-pi, such as freedom from parallel aliasing and race-hazards. A programmer using C libraries in occam-pi may have to keep track of C-specific state between calls to foreign functions, which, if not used carefully, may result in unwanted race conditions. Furthermore, interfaces which translate C's imperative programming paradigm to occam-pi's process-oriented paradigm must be created to leverage all of occam's benefits. The following sections describe scenarios where such situations can arise, and discusses how incompatibilities in the models can be approached.

3.9.1 Ensuring ordering of foreign function calls

When running occam-pi in a multi-threaded environment such as CCSP, care needs to be taken when using foreign libraries as these may rely on thread-local state. Using

a library that has thread-local state from multiple threads may result in unexpected behaviour and pose a race hazard. The SWIG-generated language bindings to OpenGL are used to demonstrate such a scenario.

It is possible that an occam-pi program that makes calls to OpenGL from multiple occam-pi, or operating system processes will result in incorrect behaviour. OpenGL is a state-machine, and many library calls will behave unexpectedly if the machine is not in the correct state. For example, OpenGL's `glBegin()`, `glEnd()` commands denote the start and finish of a rendering block. Rendering commands in such a block, as shown in Listing 3.17, must be executed in the correct order. Commands specifying vertex locations, such as `glVertex3f`, used outside of a start and end block, or called in the wrong order, may lead to incorrect rendering of items and result in visual artefacts. In the example below, vertexes specified using `glVertex3f` are joined together by lines, drawing two lines connecting three points, resulting in a right angled line. If called in the wrong order, or if other rendering commands where inserted while processing an different image would appear. It is up to the programmer to ensure that any operations that rely on OpenGL's state are serialised correctly with other processes that render graphics to the screen.

```
SEQ
  glBegin(GL.LINE)
  glVertex3f(1.0, 1.0, 2.0)
  glVertex3f(2.0, 1.0, 2.0)
  glVertex3f(2.0, 2.0, 2.0)
  glEnd()
```

Listing 3.17: A section of OpenGL code that needs to be executed serially.

In order to help ensure that this does not occur, a GUI framework based on the ideas discussed in the GTKoccam and WGoccam frameworks [Sto03, Whi00] could help enforce order on rendering commands so that they do not execute when OpenGL is in the wrong state, or in the middle of accepting rendering commands from another process. In the context of libraries that are not POSIX thread safe, a rendering manager could ensure that a graphics library is only called from the same operating system

thread. This could work providing the programmer is careful not call to use the graphics library directly, outside of such a framework.

For example, a 'drawing manager' process could control the rendering order using a barrier. Processes needing to draw to the screen would only do so while subscribed to the barrier. Since only one process would be allowed to do so at a time a drawing order would be enforced, ensuring thread safety. Alternatively, a "rendering token" could be passed to processes wishing to render, much like in a token ring network, only permitting one process to execute drawing commands at a time.

It is possible to build a drawing manager of this type into the Transterpreter, or CCSP, as part of its execution environment. By defining a new standardised top level interface[5] for graphical applications, all drawing commands could be serialised by ensuring that a programmer only uses such an interface for drawing to an application. For example, channels can be added to the top level process which give a programmer access to a drawing canvas, or a configuration channel which configures the windowing system appropriately.

3.9.2 Mapping external events to occam-pi channels

A single event-queue model is commonly used in C libraries, similar to Java's Event Queue. This single event model needs to be adapted to occam's distributed event model in order to provide an interface that is more suited to occam-pi's programming methodology. Such an abstraction can use occam-pi channels as a mechanism to notify processes of events. For example, the SDL library intercepts input and system events on behalf of the programmer and adds them to an event queue which a program is expected to read periodically to check for new events. To map these events onto a set of channels, a process can be written that reads these events and translates them into channel communications for the rest of the application to use.

Two options are viable in this scenario. It is possible to create channels which map

[5]The interface of the entry process in an occam program.

to each event type, or a single, shared channel with a defined `CASE` protocol that specifies all the possible events that can occur. In either case, an event monitor process needs to be polling continually for new events and taking them off the event queue and sending them on. Listing 3.18 shows a simplified event monitor which checks for and passes on keyboard and mouse events and the UNIX `SIGTERM`[6] signal. All other events are discarded. The listing splits events of different types into separate channels. Figure 4 illustrates how the event mechanism is implemented. While this code uses polling, which is undesirable, it is the only way to interface with SDL, barring a rewrite of SDL's event handling code.

This approach brings some language specific caveats with it. As in C, care needs to be taken to ensure that the single-event queue, and the channels leading away from this queue into the occam application, is serviced regularly, so that the event backlog does not become too large. Where the event queue cannot be serviced regularly, it is possible to create event buffers, or depending on the application (such as an interactive game) use overwriting buffers to ensure that the event queue does not get blocked for too long.

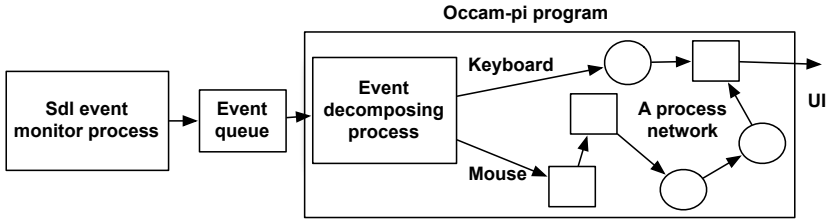

Figure 4: Interacting with events using occam-pi and SDL. Events are taken off the queue if available and sent to the remainder of the application.

3.9.3 Working with libraries that rely on callbacks

Certain C libraries implement their own threading solutions, and use a callback mechanism to execute user code. Special care needs to be taken when attempting to use such libraries with occam-pi programs as the two models of concurrency are not compatible. For example, a library that provides sound support may run a separate buffering

[6] `SIGTERM is aliased by SDL as SDL.QUIT`, as seen in Listing 3.18

process in parallel to the programs main thread of execution in order to ensure sound events do not get blocked by prolonged computation in a program's main loop, thereby ensuring continuous playback. As it is not possible for a C program to modify the occam-pi stack directly, a C function must be written so that its pointer can be passed as a parameter for the callback.

```
PROC sdl.event.monitor(CHAN BYTE keyboard!,
                       CHAN [2]INT mouse!,
                       CHAN BOOL quit!)
  SDL.Event event:
  INT result, type:
  INITIAL BOOL run IS TRUE:
  SEQ
    WHILE run
      SEQ
        SDL.PollEvent(result, event)
        IF
          result > 0
            SEQ
              get.SDL.Event.type(type, event)
              IF
                type = SDL.KEYDOWN
                  keyboard ! get.keystroke.value(event, key)
                type = SDL.MOUSEBUTTONDOWN
                  mouse ! get.mouse.position(event, position)
                type = SDL.QUIT
                  SEQ
                    run := FALSE
                    quit ! TRUE
                TRUE
                  SKIP
          TRUE
            SKIP
        RESCHEDULE()
    delete.SDL.Event(event)
:
```

Listing 3.18: An event monitor using SWIG and occam-pi

This example examines the SDLSound library as it uses a callback model. When programming using the SDLSound library in C, the programmer must provide a pointer to a function which will be used as a callback. This function has to accept the following three parameters; a pointer to sound sample being played, a pointer to the sound buffer,

and the size of the sound buffer. The function is called periodically by the sound process and is responsible for copying data into the sound buffer which the sound process uses to play the sound back asynchronously. The function also needs to detect when a sound sample is about to end and has to notify the main program that playback is about to complete. When programming in C, the SDLSound manual recommends setting a status flag to indicate that the sound has completed playing. The sound process ensures that it does not use the sound buffer while the callback function is being called. The flag must be set atomically so as not to pose a race hazard, because the flag is going to be accessed concurrently. Providing the flag variable is aligned to the architectures word-boundary, most modern architectures guarantee atomic writes to memory for word-sized data, preventing the risk of word-tearing. Therefore it is sufficient to declare that the flag variable is **`volatile`**, to indicate that the variable must always be fetched from memory each time it is accessed to ensure that an up-to-date copy is used.

Two approaches can be used to map the use of callbacks to occam-pi's process-oriented model. In both cases, a callback function is still required. The first option, is to use a polling loop written in occam-pi that mimics the C solution described above. Polling, however is considered to be inefficient and error prone, especially if other alternatives are available. The second, more favourable, option is to provide a mechanism for a C callback to communicate with an occam-pi process over an external channel.

The latter approach was first used in [Sto03], where a C interface to CCSP permitted a C function to manipulate a channel end correctly by invoking a specially crafted C function which instructed the CCSP runtime that a channel communication had occurred. The first use of this approach, however, was not thread safe and would not work in a parallel system. The newer CIF interface, formalised in [Bar05], allows specially constructed C functions to behave like occam-pi processes and the use of most of occam-pi's safe communication and synchronisation facilities. Using CIF it is possible to obtain a reference to a channel end and store it in C's global memory. It is, therefore, possible to write a callback function that uses a previously initialised reference to a channel to send events to an occam-pi process. Such a solution still poses a risk of incorrect parallel usage, should the callback function be invoked in parallel by the

sound process. It could, however be "safe" providing that the sound process guarantees that it will not invoke the callback multiple times in parallel. In conjunction with such a guarantee, and a parallel-aware implementation of CIF, such a system would provide an elegant solution to an inelegant problem. In the event that a library did not provide such a guarantee, the callback function would have to implement a locking mechanism to prevent parallel usage of a channel. Locking would also be necessary in the polling solution proposed above in order to prevent parallel writes to the flag variable.

It is worth noting that CIF is not yet supported on the Transterpreter, so when constructing cross-platform solutions, or using non IA32 platforms the polling mechanism is the only viable option for the time being. A CIF compatible mechanism is planned for the Transterpreter and is discussed further in Chapter 7.

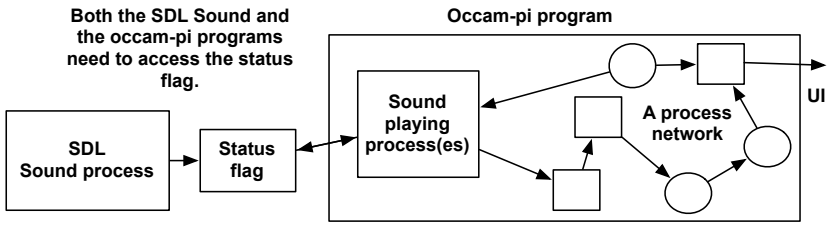

Figure 5: Interacting with callbacks using occam-pi. The status variable must either be polled from occam-pi or the callback function must be instrumented to use CIF.

3.10 Chapter contributions and summary

The SWIG occam-pi module makes it easier to build rich applications in occam-pi. It makes the language more amenable to designing and developing reactive concurrent applications which interface with the host system. Evidence for this can be found in the number of research and teaching projects that now rely on bindings created using SWIG occam-pi as described below. The following sections also identify several areas where the SWIG occam-pi module can, and will, be improved and extended.

3.10.1 Projects that made use of SWIG occam-pi

SWIG occam-pi has helped create a number of bindings to libraries and applications which has made a number of occam-pi-based projects possible.

An interface to the Player/Stage [BCG+05] robotics simulation and control environment was developed using the occam-pi module for SWIG to enable occam-pi to control both simulated robots and real-world platforms such as the Pioneer III [Mob09]. The ability to generate bindings to this library automatically enabled concurrency teaching and research using occam-pi as a control language. Furthermore, upgrading the bindings to new versions of the libraries proved to be simple, allowing new features in the library to be exploited without a substantial time-investment.

As a result of the availability of Player/Stage bindings, Robodeb [DJJS], a Debian Linux based concurrency oriented robotics programming environment for teaching has been developed. The freely available environment can be run using the free VMWare Player or VMWare compatible virtualisation programs and includes the Player/Stage environment, the Transterpreter virtual machine, and JEdit, a fully featured IDE that supports occam-pi. The environment allowed students to start programming robotic behaviour using concurrency as an abstraction of the real world with very little effort. Programs created by students in this environment are also usable on real world robotics platforms such as the Pioneer. Robodeb and the use of occam-pi in robotics for education and research are described in more detail in Section 6.1 on page 119.

Sampson's Game of Life implementation in occam-pi, using the OpenGL library is described in [SWB05]. Figure 3 on page 50 is an image of the running application, depicting a cellular automaton written in occam-pi. Sampson's Occade [Sam09a, Sam09b], an arcade game development framework for occam-pi, was created for use in the University of Kent's concurrent programming course. Occade is an extention of a simulation environment used for modelling complex systems as part of the CoSMoS Project [WSST08]. CoSMoS is also investigating the modelling of robotic swarms by making use of the Player/Stage bindings for occam-pi.

Finally, wrappers to OpenMPI [ANL09, BDV94] compatible libraries have been

made enabling the Transterpreter to be used in a networked environment. A library for vector processing has also been made available with the aid of SWIG, enabling the use of the vector processing hardware commonly available on desktop and server processors. This is described in more detail in Chapter 5

3.10.2 Future Improvements to the SWIG Module

A number of improvements and extensions are planned for the SWIG module. It be desirable for SWIG to be able to generate code for functions using typeless pointers automatically, so that they can be passed a range of data types. Further investigation of the SWIG macro system, which implements a similar feature for `malloc` and `free`, would be needed.

While the initial goal of the occam-pi module was to support the wrapping of C libraries, there is nothing preventing it from supporting the wrapping of existing C++ code. SWIG is capable of parsing and generating C code and target language wrappers for most C++ code. A future extension to the SWIG occam-pi module would be to add support for wrapping C++ classes. Some work has already been done to support the wrapping of C++ code in the occam-pi module for SWIG, but it is not yet complete. Completion of support for C++ would result in a further increase to the code base that is currently accessible for occam-pi programmers.

3.10.3 SWIG and CIF

The occam-pi C-interface, CIF [Bar05], could help extend SWIG built interfaces. Since C code can be embedded into SWIG interface files, it is desirable to extend SWIG to support annotations for callback functions, or global variables and shared data to provide an automatically generated, and safe(er) interface to these. For example, libraries which make use of callbacks can have customised **PROC**s written in CIF which are run on program start-up. The **PROC**(s) can have a channel interface which can notify an occam-pi **PROC** of callback events such as keyboard and mouse presses. Exploring the possibility of embedding CIF into SWIG wrappers is part of the planned future work.

Chapter 4

occam-pi on the Cell BE

Multi-core processors are becoming commonplace in desktop computers, laptops and games consoles [Rat05, BDK+05, KAO05], and this trend is being steadily pushed to 10s of cores by industry leaders such as IBM, NVidia, Sun Microsystems and Intel [KAO05, LNOM08, Van08, SCS+09]. Traditionally, the programming of such concurrent systems has been considered difficult and unreliable when using languages that lack compiler support for concurrency [Lee06, Boe05]. The Cell Broadband Engine, and its successor the PowerXCell 8i, generally referred to as the Cell processor, are examples of a readily available and affordable multi-core processor.

The Cell processor contains two different processor types, and nine independent processor segments. Programs wishing to exploit all processors have to be written as two or more separate programs due to the difference in architecture between the processors. A master program is compiled for and executes on a PowerPC compatible "master processor" and is used to launch and control the programs compiled for the SPU's. The SPU's are the Cell's secondary processors which provide most of the Cell's computational power. These separate programs must be able to synchronise and share data. Combined with the need for explicit memory management on the SPU, as these have their own dedicated memory spaces, makes software development on the Cell more difficult than programming traditional processors. The Cell requires a different approach than the currently used threading models to software development if it is to be used successfully. This chapter describes a new approach to programming the Cell using the

occam-pi programming language. It also goes on to compare existing approaches to software development on the Cell with the proposed occam-pi solution, along with details on how the Cell processor works and what features it has.

4.1 An Overview of the Cell Broadband Engine

The Cell consists of a PowerPC [RSJ04] core and up to eight SPUs [Fla05], vector processing units, connected via a high speed bus on a single chip. The Cell processor is widely available in high-end blade servers from IBM, as add-in PCI Express cards or, more cost effectively, in Sony's Playstation 3, or for free in the form of cycle accurate simulator provided by IBM [IBM06]. This section presents background information on the Cell architecture which should help clarify some of the Transtepreters implementation details given in Section 4.3.

4.1.1 PowerPC Core (PPC)

The Cell's PowerPC core was designed to be binary compatible with existing POWER architecture based processors, allowing the execution of pre-existing binaries on the Cell's PPC core without modification. The core has built-in hardware support for simultaneous multi-threading, allowing two threads to execute in parallel on the processor. An AltiVec [DDHS00] Single Instruction Multiple Data (SIMD) vector processing unit is also present. One of the main differences between the Cell's PPU and other POWER based processors is the lack of a branch prediction unit (resulting in a smaller/simpler processor), and some of the more esoteric POWER instructions, which are implemented in software (microcode) on-chip.

4.1.2 Synergistic Processing Units (SPU)

The SPU processors are dedicated vector processing units. They are what allows the Cell BE to achieve its high level of computational performance, and are also the most interesting part of the processor from a programming perspective. They were designed

to provide optimal 32-bit floating point vector calculations and lack the scalar and integer instructions commonly found on main system processors. The much simplified ISA results in a physically small, low power, processor that compromises general purpose performance for high performance vector processing. While the SPU is still able to execute general purpose code, and this does not require specialised coding, this will not make good use of processor resources. Optimal coding for the Cell will make explicit use of the processor's vector processing units.

Each SPU is equipped with a very high speed 256KB local store for program and data, and each unit has a dedicated built-in DMA (Direct Memory Access) controller. The memory controller, and hence memory access, is programmed explicitly from the SPU to manage data flow between system memory and the local store. The memory controllers are able to DMA chunks of data up to 16KB in size to and from SPU local stores from system memory without interrupting computation. Larger chunks of data can be queued into multiple DMA requests up to a maximum of 128KB at any given time.

The memory controllers also co-ordinate most of the inter-processor synchronisation. Synchronisation and communication are achieved by reading from and writing to special-purpose memory-mapped registers designated for this task. Each SPU is equipped with a set of three 32-bit mailboxes (registers) for communication/synchronisation with the PPC. Two are outbound, blocking mailboxes, one of which interrupts the PPC when a message is sent. The third, inbound mailbox, is for receiving messages from the PPC. Each SPU is also equipped with two inbound 32-bit registers, called `SigNotify1` and `SigNotify2`, which any SPU or PPC can write to, with either overwriting or blocking behaviours.

The SPU has 128 general purpose registers that are each 128-bits wide. The SPU instruction set is designed to operate on these registers as quads of 32-bits, and has practically no instructions for scalar operations. Instead, when a scalar operation is required, in many cases data must be masked out of a 128-bit segment in memory, operated on, and then copied back to the local store with another masking operation. It is possible to designate scalar variables that occupy an entire 128-bit segment of

memory to avoid the need for masking — this can provide a significant performance boost for scalars that are accessed often at the cost of some wasted memory.

In line with the processors small design, the SPU also has no branch prediction hardware which is common in larger processors that have long instruction pipelines. The SPU also has a relatively deep instruction pipeline resulting in equally high penalties when branches are missed in code because the instruction stream needs to be flushed and refilled with the appropriate code before the processor can continue computation. Therefore when writing code for the SPU it is better to avoid branching where possible.

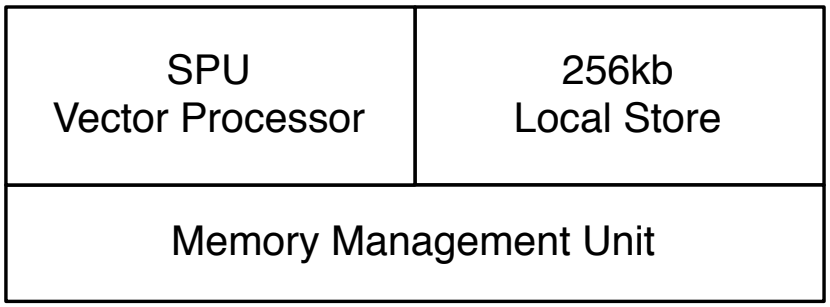

Figure 6: A diagram of an SPU.

4.1.3 The Cell's Element Interconnect Bus

A key component of the Cell Processor is the high-speed Element Interconnect Bus (EIB) through which all of the processing units and main memory are connected. The EIB is an on-chip bus which allows all the processing units to communicate with each other directly with very low latency because it avoids having to access main memory. A diagram of the Cell processor and how the Cell interconnects all of the elements can be seen in Figure 7.

4.2 Current approaches to programming the Cell

A number of approaches to simplifying or automating parallelisation of programs for the Cell are being developed. The approaches can be categorized into automated parallelisation, message passing approaches or approaches which use a virtual machine or

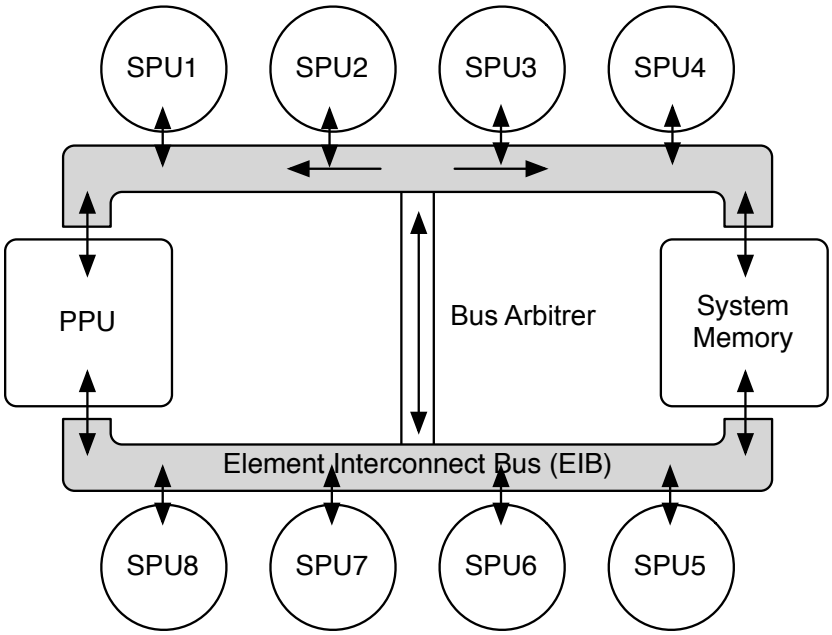

Figure 7: A diagram of the Cell BE.

scheduling library to implement lightweight multi-threading on the SPU and provide facilities for inter processor communication.

Programming the Cell is currently mostly restricted to C, C++ or Fortran due to the availability of compilers for the SPU. More recently there have been attempts at creating a native JVM for the Cell [WNGG, NF08], but in most cases, use of the SPU's from most languages is restricted to writing small C or C++ programs that are launched through calls to C libraries for the Cell [IBM05]. This is partly because C and C++'s native compilation and support for vector operations and direct access to the memory management units is able to provide superior support for the SPU than other languages currently have. Popular languages, that rely on virtual machines for execution also have memory requirements that exceed the available memory on an SPU and are therefore more limited on the Cell.

This section presents an overview of programming language implementations available for the Cell. It is not intended to be exhaustive, however, as only the implementations relevant to this Chapter are included. Commonly used programming paradigms are discussed and their Cell-specific implementations are described. The languages and

techniques described are predominantly constrained to C/C++ and CSP related languages. Of the solutions presented, OpenMP, Microtasks for MPI and CellSs all have a form of automated call graph dependency tracking which is used to determine the order of execution for the tasks that they generate. One of the key optimisation factors for these solutions is ensuring that their call dependencies are worked out efficiently, so that swapping is reduced.

4.2.1 OpenMP

Probably the simplest approach to parallelising existing programs is to use a compiler that supports OpenMP [DM98] on the Cell such as IBM's commercial XLC [IBM09] compiler. This approach is commonly used for parallelisation of existing, single-threaded code and is available for C, C++ and Fortran. It allows a programmer to exploit the concurrency that is inherent in loops to split CPU-intensive portions of programs into processes for execution across multiple processors through the use of preprocessor directives that instruct the compiler about safe transformations to generate parallel code. Some complexity of parallelisation is pushed into the compiler, making it do the harder work of generating the appropriate SMP code for a given section of code. This process is not automated and the programmer has to supply the compiler with hints as to what a programs variables relations and dependencies are. Furthermore, the compiler is also unable to tell if a programmer has correctly annotated their code.

To denote a section of code that is to be execute in parallel the `omp parallel` directive is used. For example, Listing 4.1 shows an `omp parallel` block where variables `var1` and `var2` are thread-local and `var3` is a shared variable. A typical OpenMP program will follow an execution pattern similar to the one illustrated in Figure 8.

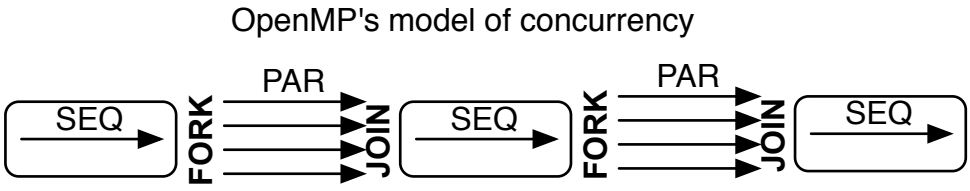

Figure 8: A model of how concurrency is achieved using OpenMP.

```
#pragma omp parallel private(var1, var2) shared(var3)
  {
  Parallel section executed by all threads
      ...
  All threads join master thread and disband
  }
```

Listing 4.1: OpenMP fork-join parallel example

Using OpenMP however, it is easy to specify a directive incorrectly, leading to race conditions or incorrect parallel usage as the following example demonstrates. For example, in order to instruct the compiler to split a **`for`** loop into `N` concurrent processes a parallel section is annotated with a **`#pragma`** as shown in Listing 4.2. A **`for`** `num_threads(N)` directive inserted in front of a **`for`** loop will result in the compiler generating code that will start a team of `N` processes, sharing the work of the **`for`** loop. The programs main thread of execution will only resume once all processes complete, due to the implicit barrier at the end of each `omp parallel` block.

Listing 4.2 demonstrates the use of OpenMP's `parallel` **`for`** `num_threads` directive that will result in a race condition, due to parallel access of the variable `shared`. The listing will compile without warning because the compiler alone will not be able to determine that parallel access is occurring and will result in the variable `shared` having an incorrect value once the threads complete. While it is possible to specify which variables are shared in an OpenMP block as shown in Listing 4.3, in this case by specifying the `reduction` keyword and informing OpenMP to sum the combined results `shared` once the parallel section ends, it is up to the programmer to ensure correctness of any parallel sections that have been written and the compiler is of little help when it comes to verifying the correctness of parallel programs created using OpenMP. While there are tools which attempt to assist the programmer in writing correct code using OpenMP such as the C/C++ only, commercially available, VivaMP [Sys09], there is no guarantee that such a tool will be able to catch all cases of incorrect parallel usage because of the lack of formal specification for concurrency in the C language.

```
double IncorrectOpenMP(int a)
{
  double shared=0;
  #pragma omp parallel for num_threads(2)
  for (int j=1; j<=a; j++) {
    shared += j * a;
  };
  return shared;
}
```

Listing 4.2: Race hazard example using OpenMP.

```
double CorrectOpenMP(int a)
{
  double shared=0;
  #pragma omp parallel for num_threads(2) \
    reduction(+: shared)
  for (int j=1; j<=a; j++) {
    shared += j * a;
  };
  return shared;
}
```

Listing 4.3: Race condition free OpenMP example.

OpenMP also allows more fine-grained control over program distribution through use of the `omp section` directive. The directive is used inside a `omp parallel` section and can be specified repeatedly to denote multiple parallel sections. All `omp sections` declared in an `omp parallel` block must complete in order for the program to resume its main thread of execution. The same, if not greater risks apply when using `omp section` as when using a `omp` **`for`** directive as the scope for incorrect specification of variable dependencies increases.

In practice what happens on the Cell is that a C program which has been annotated with OpenMP directives can be compiled to have the threaded sections of a program

execute on the SPUs. When an `omp parallel` section is reached, the main process suspends execution and the shared tasks are loaded on the SPUs. When there are more tasks than SPUs, unallocated tasks are queued until an SPU become free to process them. Once all tasks are completed, the main thread of execution resumes on the PPC. XLC's OpenMP implementation does not allow hand-written, SPU specific SIMD code to be used in OpenMP sections, and a programmer must rely on XLC's automatic SIMD transformations to make use of SPU's SIMD capability.

Newer versions of the IBM Cell SDK support code overlays on the SPU, where only the code is replaced, leaving the data on the processor, and bringing the executable to the data. When a user program has large binaries, it may be possible for OpenMP to replace executable code only, leaving the data on the processor, both allowing larger binaries to be used, and reducing some of the latency resulting from swapping.

4.2.2 The Octopiler

IBM's Octopiler [Eic06] is an auto-parallelising compiler for the Cell based on the XLC compiler and it builds on and extends OpenMP concepts. In addition to providing support for OpenMP, IBM have added additional features to the compiler which allow for further automation and more involved transformations of existing code. These include compiler optimised partitioning for data and code to run on both the PPC and SPUs in the system, automatic generation of SIMD code through loop unrolling, and other specialised optimisations for processor elements in the Cell architecture.

Unfortunately not all code can be modified or annotated easily to gain performance advantages from auto-parallelising compilers. Alternative methods to automatic parallelisation and OpenMP could provide the ability to make use of the Cell without requiring expert programming.

4.2.3 Stream Programming for the Cell

Stream processing is a programming model that allows amenable applications to take advantage of a limited form of parallel processing by taking advantage of SIMD compatible data-parallelism. A stream — an array of data — can be automatically split across multiple computational units, such as the processing units on a GPU or in the Cell's case, its SPUs. Streams are programmed without explicit memory allocation, synchronization, or communication between processing units. A stream program will generally consist of a software kernel, that describe the operation to be performed on one or more arrays of data, and where the result should be stored. A procedural program, which contains all the program logic, will invoke stream-kernels to perform computation on arrays. Kernels can be chained together, combining a set of operations that a programmer may want to perform on a set of data, resulting in a "stream of computation". Stream programming is reminiscent of functional programming due to its semantics, where a kernel (function) is applied to a stream (list) recursively.

Stream programming languages include Brook [BFH+04], CellSs [PBBL07] and OpenCL [Gro08]. Of these only CellSs has a freely available implementation for the Cell, but the concepts are generally applicable to an architecture like the Cell. CellSs uses preprocessor directives to extend C, in a way not dissimilar to OpenMP directives, in order to annotate code sections that can benefit from vector parallisation. A sample stream kernel which calculates factorial is defined using CellSs as shown in Listing 4.4. The `css task` directive informs the compiler that `n` is the input parameter for this kernel and `result` will hold the output values. Because the `task` pragma syntax is closely bound to the functions parameter list, it is quite easy to ensure that all the input and output variables are well defined.

Invocation of the kernel occurs inside a `css start` and `css end` block. To compute the factorial for each member of a list of values the `factorial` function is invoked from inside a loop. The size of a work unit is defined by the programmer, in the example, a unit consists of 3 values. The factorial kernels are loaded onto two SPU's and executed. An implicit barrier synchronises the running kernels on completion.

```
#DEFINE UNIT 3
#pragma css task input(n) output(result)
void factorial(unsigned int n[UNIT], unsigned int result[UNIT]) {
  for (int i = 0; i < UNIT; i++) {
    result[i] = 1;
    for (; n > 1; n--) {
      result[i] = result[i] * n;
    }
  }
}
```

Listing 4.4: A sample factorial function in CellSs taken from [PBBL07].

```
int factorme[SIZE] = [128, 256, 362, 411, 501, 666];
#pragma css start
for (int i = 0; i < SIZE; i += UNIT) {
  factorial(factorme[i], result[i]);
}
#pragma css end
```

Listing 4.5: Invoking a CellSs kernel.

The primary difference between CellSs and OpenMP is that CellSs allows for a more fine-grained control of work load distribution. When using the `css task` syntax, which is comparable to OpenMP's **`for`** sytax, it is possible to manually determine the work-size of each segment. In the example above, the workload is not evenly distributed, as the computation of a factorial value for larger numbers is more compute intensive than for smaller values. Where OpenMP automatically partitions work, CellSs permits the programmer to specify how sets of data should be partitioned across processes/streams. Therefore, it is possible to rewrite the example above to balance the work better between SPUs by, for example, allocating the first four values to the first SPU, and the last two to the second. Finally, CellSs permits the use of SPU intrinsics inside kernel functions, giving a programmer explicit control over SIMD level parallelisation on the SPUs.

A nifty feature of CellSs is that it has a built in trace function, which measures DMA execution. This makes it easier to measure the amount of DMA calls that result from running a program, and optimise it accordingly.

4.2.4 MPI Microtasks for the Cell

A subset of the MPI specification [ANL09, SO98] has been implemented for the Cell Processor, known as MPI Microtask programming [OIS+06]. MPI is often the framework/API of choice used for developing software on distributed and multi-core systems. The MPI Microtask compiler converts, through the use of a pre-processor, an MPI program into many small tasklets. Microtasks are defined by communication at the start and end of a task, with any computation being performed between communications. As there will typically be more microtasks than SPUs, a static, compile-time defined scheduler performs context switching of the microtasks at runtime as it is not possible for more than one task to execute on any given SPU at a time. As the overhead of context switching for SPUs is quite high, the compiler attempts to determine communication dependencies at compile-time to minimise the number of context switches required in a program. A program compiled using this technique will produce many small computational kernels resulting in a system that resembles the stream programming model, but is different as taskelets may contain logic which decides where the results of a computation are communicated to next.

Programs written using MPI Microtask may result in many SPU context switches as there is no mechanism for interleaving computation from multiple kernels on an SPU. The overheads of context switching may result in significant performance penalties for programs that communicate arbitrarily with many microtasks. Furthermore no provisions are made for preventing or detecting deadlock. While race hazards can be avoided through careful programming, it is entirely up to the user to ensure this.

4.2.5 Trancell — an occam-pi translator for the Cell

A prototype in-place replacement for tranx86, targeting the Cell Trancell [JS07] which can produce SPU binary code directly from ETC assembly was created in 2007. This translator allowed a small subset of occam-pi to compile and run on a set of SPUs. Interprocessor communication was performed by sending messages to the PPU which routed messages to their destinations.

The developers of Trancell concluded that channel communication implemented by using the PPU as a router quickly became a bottleneck for interprocessor communication as messages could only be passed 32-bits at a time. Furthermore, the complexities of code translation and debugging on the Cell proved difficult and the project has since been abandoned.

4.2.6 CellCSP

A more recent development is the CellCSP library. CellCSP is a C++ library developed by Kristiansen [Kri09, Kri08] which exposes a CSP-inspired API for the Cell, featuring co-operative scheduling of CSP processes, support for channel communication and alternation. In practice, the library loads small C kernels on each SPU. These provide facilities for loading and storing processes from main memory, as well as handling communication. The kernels permit up to two concurrent processes to execute on each SPU. Processes are co-operatively scheduled across all the SPUs, and are switched out of the SPUs into main memory by a co-operative scheduler running on the PPU. The PPU sends scheduling instructions to the SPUs using the Cell's mailbox registers. The system attempts to mitigate DMA-latency, caused by switching processes in and out of main memory, by permitting continued execution of a second process during the switch, thus ensuring SPUs are not left idle.

The library is still considered a prototype by its author. Kristiansen's initial focus was on creating a scheduling mechanism for the SPE-cores. Kristiansen also concludes that the library has scope for improvement in terms of process scheduling efficiency and suggests using system memory for the purpose of scheduling processes.

4.2.7 Bandwidth/Optimisation Techniques on the Cell

One of the key issues when programming the Cell is ensuring that processing time to data ratio is kept high — programs optimised for the Cell, much like other multi-core processors, need to be bandwidth-aware. The only way to achieve and sustain high percentages of processor occupation is to ensure the memory bus is not overloaded so that processors do not spend a large proportion of time waiting for data. Ideally programs can be written so that results of a computation or new data are transferred between the SPU local store and system memory while the SPU is carrying out computation on other sets of data.

Further optimisations can be made by avoiding the use of conditional instructions. For example, loops which can be "unrolled" can perform much better than a looped version of the same code. The downside of this optimisation is that it leads to larger binaries taking away some of the already limited space on the SPU.

Where possible all data should be stored as and operated on as 128-bit aligned quad-words, by making use of the SPU native SIMD instructions. Scalar values, that are used regularly, should be stored in their own quad-words (not using the remaining 96-bits) to avoid the need for masking individual values in and out of a quad word before operating on it. As with loop unrolling, this optimisation comes at the expense of SPU local store space.

Finally, when making larger transfers data should be copied in 16KB segments between the local store and system memory, as this is the maximum size available for each DMA request. For smaller data-sets, smaller amounts should be used, as these will copy faster. Larger copy operations should be broken into 16KB segments where appropriate.

When only small amounts of data, such as individual 32-bit values need transferring, the SPU mailboxes may be used. Some care needs to be taken, as it is possible to overwrite these prior to reading. A special non-overwriting mode must be set for the mailboxes to prevent this from occurring.

4.3 The Transterpreter on the Cell Broadband Engine

The Cell port of the Transterpreter addresses architectural issues, such as program distribution, and support for safe interprocessor data synchronisation and communication. The Transterpreter, being a highly portable interpreter for occam-pi, provides a means for running occam-pi on a new platform in a short amount of time. This circumvents the need to write, and later maintain, a separate compiler back-end for the platform. Furthermore the port of the Transterpreter to the Cell is used to explore occam-pi in the context of the Cell processor to gain an understanding of what would be required in order to port the language to the Cell processor using the Tock compiler.

The core of the Transterpreter is portable across platforms, it has no external dependencies and it builds using any ANSI compliant C compiler. For the Transterpreter runtime to be useful on a given architecture, a platform specific wrapper needs to be written which provides an interface to the underlying hardware. In the case of the Cell two separate wrappers where needed, one for the PPC core, and one for the SPU.

While porting the Transterpreter to the Cell two mechanisms, with differing performance characteristics depending on the scenario,for channel communication were developed. Both approaches maintain occam-pi semantics while relying on different features of the underlying hardware. As the underlying hardware does not map directly to the semantics of CSP/occam-pi channels, software to manage channel communication needed to be implemented. Furthermore, the Transterpreter scheduler needed to be extended to support the handling of interrupts generated by the Cell such as those generated by the SPU time functions.

4.3.1 Program Distribution

The Transterpreter for the Cell consists of customised PPC and SPU executables. The PPU Transterpreter is programmed to load SPU Transterpreter executables onto the SPU when it is run. Once the PPC and the SPUs are all running, the program bytecode designated for execution is loaded into main memory. A pointer to the program bytecode is passed to all the SPUs which then copy the program bytecode into their local

stores from system memory. Once the copy is complete all the Transterpreters synchronise and begin executing the byte code. Currently all Transterpreter instances execute the same bytecode. Given this setup, in order for program flow to differ on processors, a program must query the Transterpreter about its location in order to determine which processes to execute. A programs location can be determined by the unique CPU ID, a number from 0 to 8; Zero is assigned the PPU and 1-8 to the SPUs. Each Transterpreter instance gets its CPU ID assigned at start up.

4.3.2 Inter-Processor Communication

occam-pi channels are unidirectional, blocking, and point-to-point. The individual SPUs of a Cell processor are not so restricted in their communications; therefore, both the compiler and the wrappers must provide support for making the mapping from occam-pi to the Cell hardware consistent and safe.

The blocking nature of channel communications provides explicit synchronisation points in a program. While the compiler provides checks for correct directional usage of channels when compiling, the Transterpreter wrappers must ensure such that channel communications between processors are blocking and unbuffered. Two methods of communication are presented using different approaches and hardware.

4.3.3 Communication using specialised registers

The first method uses specialised registers on the Cell which enable the sending of small messages between processors over the on-chip EIB bus. Communication options using these registers are described here.

SPU to PPC Communication

The SPU-to-PPC mailbox registers are 32-bit, unidirectional, non-overwriting buffers. When empty, a mailbox can be written to and execution can continue without waiting for the mailbox to be read. When a mailbox is read, it is emptied automatically. When a mailbox is full, the writing process will stall until the previous message has been read.

The SPU can receive soft interrupts when one of its mailboxes is read from or written to.

The mailbox registers are used to implement channel communications in occam-pi between the PPC and the SPU. In order to preserve the channel semantics of occam-pi, a writing process is taken off the run queue and set to wait for the "mailbox outbound read" interrupt to occur. The communication only completes when the mailbox is read by the PPC. The SPU is able to pass short messages quickly by continuously writing to the mailbox while the PPC continuously reads. The PPC does not receive interrupts when its outbound message is read and must poll the mailbox to check if the read has occurred.

Inter-SPU Communication using SigNotify

For SPU-to-SPU communications two inbound registers are available on each SPU, named `SigNotify1` and `SigNotify2`. The registers can be programatically configured to be non-overwriting and, like the SPU-to-PPU mailbox registers, can only be cleared by a read.

Using these registers is efficient for sending word-sized messages or synchronising SPUs. This capability is useful since it allows for communication and synchronisation without taxing the memory bus. In this case the `SigNotify2` register is used for sending data and `SigNotify1` for the sender CPU ID. Where messages are longer than 32-bits the length and type of the data being sent must be determined at compile-time. A limitation of this implementation is that safe communication is limited to pairs of processes — when a pair of processes is communicating across SPUs, no other processes residing on these SPUs should communicate with that pair as the hardware is unable to distinguish where a given message is coming from.

It would, however, be possible to interleave multiple channels using the `SigNotify` registers provided there was compiler-level support for creating such an interleave. Inter-processor channels could have additional meta-data such as global process or channel identifiers which could be communicated along with the data. This would allow the incoming/outgoing data to be matched with specific channels and permit the

runtime to multiplex these appropriately. There would be a practical limit to the number of channels that could be used via this mechanism, due to the mailboxes 32-bit size, only permitting so many channels to exist globally. Additional support would have to be added to the runtime to support encoding and decoding of channel communications to ensure that messages arrive at the correct destination.

4.3.4 Using DMA for communication

While the mailbox and `SigNotify` communication facilities are suited to small messages, it becomes more efficient to use system memory for large messages which is what the second method uses. In order to reduce contention on the word-sized registers, programs that send large messages do so by copying from local store to system memory. This removes the need for multiplexing channels on the `SigNotify` registers, as it is possible to allocate space for multiple channels in memory that do not interfere with each other. Different strategies can be used for letting other processors know when a message has been copied into memory for retrieval. For example, when a large-message copy completes, a value can be set in one of the `SigNotify` registers, providing these are not already in use, letting the destination SPU know that it has data waiting for it. Alternatively an atomically set 'ready flag' can be used in memory for controlling when messages are ready.

The current implementation uses a ready flag to notify a reader that the data has copied successfully. When the Transterpreter starts up, space is pre-allocated in system memory for sending large messages. Eight 16KB chunks of memory are allocated for each SPU, and one for the PPU for receiving data — one chunk for every other processor to send data to. Each SPU then receives a list of pointers to the memory locations that they can write to. For example, SPU 2 will receive pointers to the second 16KB chunk of memory of each SPU's/PPU receiving memory. This ensures that when two processors are sending data to a third processor, no portion of memory is overwritten unintentionally as they have separate, pre-allocated memory segments.

When a write to another SPU is initiated, the data to be sent is DMAed into the appropriate 16KB chunk of memory. Once the copy completes, a ready flag is set atomically by the writing SPU to notify the destination SPU that it is safe to read the channel's data. The writer process on the sending SPU is then moved to the interrupt queue in the scheduler and waits for the ready flag to be reset, confirming the completion of the read.

The reading SPU can detect when a process is waiting on input from an external channel and begins to poll the ready flag in system memory. Once the SPU detects that the ready flag is set, it DMAs the channel's data to its local memory and resets the flag to indicate that communication has completed.

SPU communications comparison

There are some advantages and disadvantages to the above communication methods. Direct communication between processors using mailbox registers is very fast for small messages, but becomes inefficient for exchanging larger amounts of data as the sending processor needs to stall until the receiving processor reads from, and clears, the `SigNotify` mailbox so the next message can be sent.

When using the first scheme it is difficult to send more than one 32-bit value at a time efficiently as one of the mailboxes has to be used for specifying where a message is coming from. The receiver has to assemble a message being received a word at a time as other processors could be sending messages to the same processor, resulting in messages being interleaved. Using this kind of system for large amounts of data would constrain transfer rates quickly, and using the registers becomes a bottleneck for communication.

Alternatively, it is possible to use a combination of the `SigNotify` registers and mailboxes for notifying processors when data is available to them in memory. This approach may prove to be the best solution. However, there is a risk that, with large numbers of channel communications, the registers could become a constraining factor.

As both the PPU and the SPU's are able to atomically read and write 32-bit words in system memory, the most general solution, which supports an arbitrary numbers of channels, will use flags in memory for synchronisation. This is the approach used in the

prototype presented; other channel communication methods will be evaluated in future work.

4.3.5 Scheduling

The Transterpreter uses a co-operative scheduler with three queues: the run queue, the timer list and the interrupt queue. Process rescheduling only occurs at well defined points during execution. This implies that the interrupt queue is only checked when the scheduler is active and, if interrupts are ready, processes waiting on them are made runnable by moving them from the interrupt queue to the back of the run queue.

This behaviour strongly encourages programmers to make use of occam-pi's concurrency features so as not have a processor stalling whenever a process needs to wait on an interrupt. occam-pi processes have a very low memory footprint and context switching cost which encourages the development of programs with many concurrent processes, so that a program can have processes waiting on external communication to complete overlapping with processes that are performing computation.

Should a programmer wish to write programs using a more sequential paradigm, buffer processes could be used to handle communications. Figure 9 illustrates how a writing buffer process can be used when communicating with another processor. This way, only the buffer process stalls while waiting for communication to complete, allowing other processes to continue executing. Similarly, a dedicated reading buffer process can ensure that data is read in as soon as it is available reducing potential stalls in the network and keeping all the processors busy. This can be made particularly effective by running the buffer processes at higher priority than other processes.

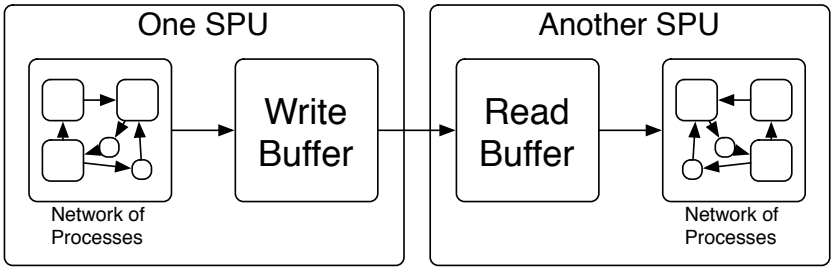

Figure 9: The read/write buffers allow computation to continue with less interruption.

4.3.6 Timers on the Cell

Measuring time on an SPUs involves setting and reading an instruction counter that decrements the value in the 'time' register at a regular interval. The interval is determined by the SPU clock speed and is converted to microseconds/seconds. When the counter reaches, or nears zero, it needs to be reset to a high value again by the runtime in order to continue measuring time correctly. This needs to be taken into account when calculating elapsed time on the SPU for the purpose of processes that are waiting on time outs.

For this to work, the scheduler must regularly check the current value of the instruction counter, and adjust its value as needed. If the scheduler is not entered for long enough, there is a danger that the counter will run out and timings will be incorrect.

More recently, IBM have released an updated API which makes using timers on the SPU much easier, through the use of a callback mechanism that is triggered on an interrupt. The API also ensures that the timer does not overrun. A future version of the Transterpreter can make use of this new functionality.

4.4 The Cell Transterpreter API

The Transterpreter exposes a built-in C-API for occam programs which is used as the basis for implementing support for higher level occam features. The Transterpreter API is tailored to support the specific needs of occam programs, and should only be used as the basis for building an occam based 'firmware' for inter-processor channel communication. The 'firmware' is dynamically loaded on startup by the Transterpreter and exposes a pure occam API to the programmer. The firmware can be updated independently of the Transterpreter, and it is possible to have different firmwares depending on the requirements. An implementation of a firmware is described in Section 4.5. The following gives an overview of the API that the Cell Transterpreter exposes to support channel communications and programming.

4.4.1 Determining execution locality

The Transterpreters that are executing on the SPUs are each assigned a unique CPU ID. A native function call mechanism in occam-pi allows the programmer to call C functions that are part of the Transterpreter. Using the native call `TVM.get.cpu.id`, the program can determine on which processor it is executing. A CPU ID value of 0 is returned if the byte code is running on the PPC and a value between 1 and 8 if it is on one of the SPUs. An example of how an occam-pi startup process on the Cell could be written is shown in Listing 4.6.

```
PROC startup(CHAN BYTE kyb, err, src)
  INT id:
  SEQ
    TVM.get.cpu.id(id)
    IF
      id = 0
        ... -- execute PPC code.
      id > 0
        ... -- execute SPU code.
:
```

Listing 4.6: An example of a start up process on the Cell Transterpreter.

Currently the only way to determine where an occam program is running on the Cell is to explicitly call the `TVM.get.cpu.id` function. The long-term plan, however, is to have the compiler generate code as above, from annotations in the code similar to the `PLACED` directives that were available on the Transputer versions of occam. This is discussed further in Chapter 7.

4.4.2 The Channel API

The Cell Transterpreter provides seven C native function calls which can be used as a basis for channel communication shown in Listing 4.7. These are in place to support the second — in memory communication method — and are not intended to be used

by developers directly, they are intended to provide an interface to the Cell on which channel interfaces can be built, as described in Section 4.5.

```
--Reading helpers
TVM.read.check(INT pid, INT check)
TVM.read.chan(INT pid, MOBILE []INT data)
TVM.read.completed(INT pid, INT check)
TVM.read.finish(INT pid)

--Writing helpers
TVM.write.check(INT pid, INT check)
TVM.write.chan(INT pid, MOBILE []INT data)
TVM.write.completed(INT pid, INT check)
```

Listing 4.7: The channel API

The two `check` functions use atomic[1] DMA commands to check the channel word, a place-holder in memory which is used to indicate whether a read from or, write to channel has occured. A non-null value indicates that data is present in the channels data block, meaning a reader can safely read the data and a writer must wait until the read has completed. A null value indicates that the data block may be written to, and is not ready for reading. The value stored in the channel word, when it is not null, indicates the size of the data segment copied into memory and lets the reader know how much data needs to be copied.

The `read.chan` and `write.chan` functions copy data between a channel's data block and SPU's local store. The `read.finish` function uses atomic DMA commands to reset the status word to a null value, indicating that a read has completed. Finally, the `completed` functions are used to query the DMA controller if a copy to or from system memory has completed. Figure 10 illustrates how a channel is laid out in system memory.

[1] On the Cell 128-bit word reads and writes are atomic.

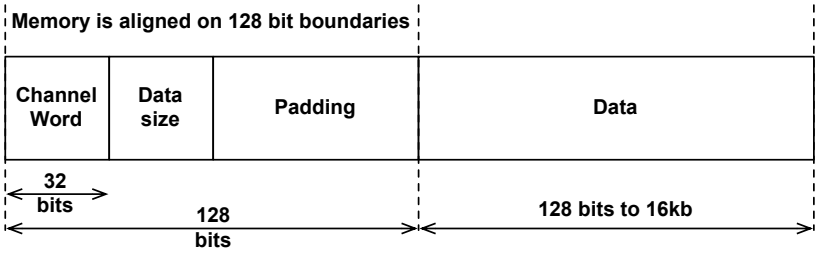

Figure 10: The layout of a channel's data in memory.

4.4.3 Using Mailboxes

It is possible to use the mailbox functions in lieu of the `read` and `write` functions described above, however the mailbox functions are limited to 32-bit message sizes and are not used in further examples. They are considered in future work as they could be used to improve channel communication performance, as such they are described here.

In order to send and receive small messages between processors, the native functions `TVM.read.mbox` and `TVM.write.mbox` are provided. The functions are designed such that their behaviour is consistent across the PPU and SPU processors. These native functions behave similarly to occam-pi channels in that they block until the communication has completed.

An example of their use is shown below in a program where nine Transterpreters are running concurrently and are connected in a ring. Each processes in the pipeline increments an incoming value and sends it on. The resulting value is printed each time it arrives back on the PPC.

The process `run.on.spu` in Listing 4.8 reads values from the previous processor in the pipeline. The value is then incremented and sent on to the next processor. The use of the modulo operator (`\`) ensures that when the value reaches the last processor in the ring, CPU ID 8, the value is sent to zero, the CPU ID of the PPC.

The process `run.on.ppu` in Listing 4.9 is executed only on the PPC core. The process header denotes that a **`CHAN BYTE`** must be passed to it as a parameter. This is a channel of type **`BYTE`**, the equivalent of a `char` in C. In this program it is connected to the screen channel that is used as a method of displaying output to the user. The "`!`"

is used to denote a write to a channel, where the RHS contains the value to be written, and the LHS denotes the name of the channel to write to. It starts propagating a value down the pipeline by writing to the first SPU. It waits for the value to complete going through the pipeline and outputs the modified value followed by a return character, by writing to the `scr` channel.

```
VAL INT stop IS 95:
VAL INT NR.SPUS IS 8:
PROC run.on.spu(VAL INT cpuid)
  INITIAL INT value IS 0:
  INT status:
  WHILE value < stop
    SEQ
      TVM.read.mbox((cpuid - 1), value, status)
      value := value + 1
      TVM.write.mbox((cpuid + 1) \ NR.SPUS, value)
:
```

Listing 4.8: A process running on an SPU.

```
PROC run.on.ppc(CHAN BYTE scr!)
  INITIAL INT value IS 65:
  INT status:
  WHILE value < stop
    SEQ
      TVM.write.mbox(1, value)
      TVM.read.mbox(8, value, status)
      scr ! (BYTE value)
      scr ! '*n'
:
```

Listing 4.9: The process which runs on the PPC and outputs data to the screen.

Listing 4.10 shows the startup process, `startup`. This gets the CPU ID using the `TVM.get.cpu.id` function and runs the appropriate process depending on the value

of `cpuid`. In this process, the `kyb`, `scr` and `err` channels are obtained 'magically' by the starting process much like command line parameters are obtained in a C `main` function. When the `run.on.ppu` process is started, the `scr` channel is passed to it as a parameter so that it can output to the screen. The `run.on.spu` process receives the CPU ID as a parameter.

```
PROC startup(CHAN BYTE kyb?, scr!, err!)
  INT cpuid:
  SEQ
    TVM.get.cpu.id(cpuid)
    IF
      cpuid = 0
        run.on.ppu(scr!)
      cpuid > 0
        run.on.spu(cpuid)
:
```

Listing 4.10: A startup process which starts other processes depending on program location.

4.5 occam-pi channel communication for the Cell

A mechanism for channel communications across multiple processors was implemented by compiling a set of occam-pi processes into the runtime. These form part of the execution environment for occam on the Cell and they expose a set of channels for inter-processor communication to the programmer. The execution environment provides one read and one write channel for every processor in the system. This allows a programmer to write programs that can communicate using occam-pi channels with processes on other processors. The complexity of inter-processor channel communication on the Cell is hidden from the programmer.

4.5.1 A microkernel for channel control

The channel implementation described here relies on the PPC portion of the Transterpreter to allocate channel words and data segments for each channel on start up. By providing, occam channels, a higher level interface for communicating with other processors it is possible to abstract over the C API used for communication.

In this instance all communication occurs via system memory, where data to be sent between processors is first copied by the sending processor from its local store into system memory, and then copied by the reading processor into its local store. Communication to and from the PPC is achieved by a read or write (`memcpy`) to system memory as the PPC has direct access to system memory, as opposed to the SPUs which must use DMA commands.

On Transterpreter start up, the microkernel creates processes to manage interprocessor communication. These processes are scheduled alongside the user program, and are connected to it through a top level interface that is exposed to the programmer. Using the Transterpreters API for the Cell, the microkernel implements channel communication that enforces occam-pi's rules on channels and allows long-running reads and writes to be interleaved with computation.

```
VAL INT NOT.PROC.P IS 0:
PROC in.chan (CHAN MOBILE []INT in!, VAL INT pid)
  MOBILE []INT tmp:
  INT in.value, complete, size:
  WHILE TRUE
    SEQ
      ASM
        LD in  -- Load the address of the 'in' channel
        LDNL 0 -- Load the status of the 'in' channel
        ST in.value -- Store the channel word in 'in.value'
      IF
        (in.value = NOT.PROC.P)  -- No proc waiting on 'in'
          RESCHEDULE()
```

```
      TRUE
        SEQ
          size, complete := 0, 1
          WHILE size = 0
            SEQ
              TVM.read.check(pid, size)
              IF
                size = 0
                  RESCHEDULE()
                TRUE
                  SEQ
                    tmp := MOBILE [size]INT
                    TVM.read.chan(pid, tmp)
                    WHILE complete <> 0
                      SEQ
                        RESCHEDULE()
                        TVM.read.completed(pid, complete)
                    in ! tmp
                    TVM.read.finish(pid)
:
```

Listing 4.11: A reading process using DMA.

Listing 4.11 shows how the reading of a channel is implemented for interprocessor communication using helper functions written in C (prefixed with `TVM`). The outer part of the **`WHILE`** loop begins with an **`ASM`**embly section that checks if a process is waiting for input on the `in` channel. This ensures that the runtime only begins polling system memory if a receiving process is awaiting input, potentially saving many accesses to system memory. If a process is waiting on input from `in`, a second loop is started which polls memory to check the availability of data. The `TVM.read.check` function polls the channel word in system memory to check for non-null values which indicates the availability of data. If the channel word contains a null pointer, no data is available for reading and the process issues the `RESCHEDULE` command, relinquishing the processor

for other processes in the queue.

If the channel word contains a non-null value, then the write has completed and the value of size indicates the size of the data in memory. An appropriately sized mobile is allocated and assigned to the temporary variable `tmp` after which the data is read into it. The `TVM.read.complete` function is used to ensure that the DMA transfer has completed. If the read does not complete immediately, the process yields using the `RESCHEDULE` command, and is scheduled for retry later. When the DMA completes, the value is sent to the waiting process on the `in` channel. When the `in` channel has been read the process continues and sets the channel word back to null by calling `TVM.read.finish` signifying that the read is complete.

```
PROC out.chan(CHAN MOBILE []INT out?, VAL INT pid)
  MOBILE []INT tmp:
  INT checkdma, checkread:
  WHILE TRUE
    checkdma, checkread := 1, 1
    out ?? tmp
      SEQ
        TVM.write.chan(pid, tmp)
        WHILE checkdma <> 0
          SEQ
            RESCHEDULE()
            TVM.write.completed(pid, checkdma)
        WHILE checkread <> 0
          SEQ
            RESCHEDULE()
            TVM.write.check(pid, checkread)
:
```

Listing 4.12: A writing process using DMA.

Listing 4.12 shows the implementation of the writing end of a channel. The `??` syntax is an extended rendezvous, which results in the writer process being blocked on

the write until the end of the **SEQ** block that begins on the following line. On receiving data on the `out` channel `TVM.write.chan` is called to initiate the DMA copy of `tmp` into system memory. `TVM.write.completed` queries the DMA controller to check if the copy has completed, and if so it sets the channel word to indicate that data is available for the remote SPU to read. Finally `TVM.write.check` is called to check if the read has completed on the other end by reading from the channel word. The `out` channel is only released once a null value is set in the channel word, and the last while loop completes. The `RESCHEDULE` commands ensure that other processes can be scheduled while the writer process polls for communication to complete.

The whole process of channel communication is illustrated in Figure 11.

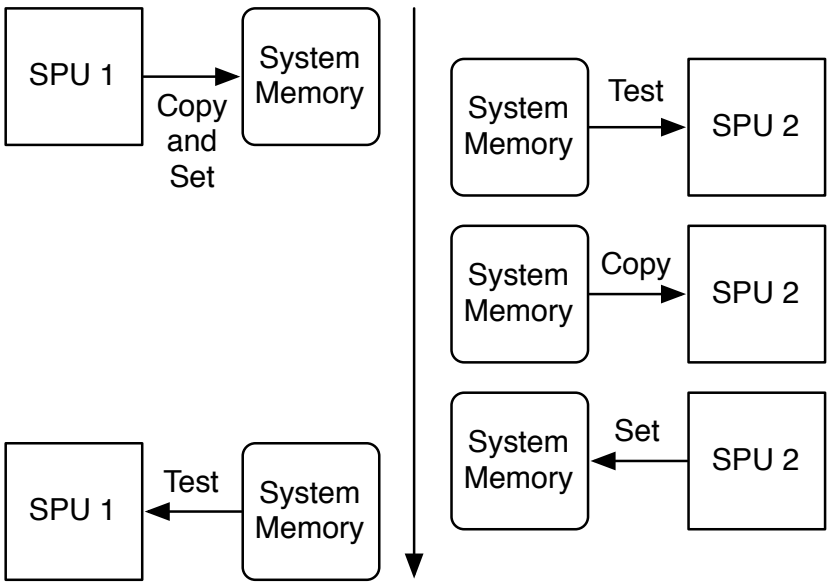

Figure 11: A visual representation of the steps taken during channel communication.

A small optimisation is possible for small packets of data, these could be packed into the second half of the 128-bit channel word when it is copied. This would reduce the number of DMA requests for each communication by two when sending one or two 32-bit words of data. This can be added to a future version of the firmware and API.

4.5.2 Limitations of the current implementation

The current implementation taxes the memory subsystem quite heavily. During communication both the reader and writer processes poll system memory to check for the channel word until reading or writing completes. While ETC assembly is used to minimise polling on the reading end, by ensuring that a reading channel is ready by checking the channels state (something that is not supported explicitly in occam) alternative solutions to polling system memory may be examined. By using the mailbox/`SigNotify` hardware available on the Cell for notification of completed reads and writes to channels or using the PPC to help manage channel communications it would be possible to reduce memory bandwidth contention.

Currently available channels between processors are statically defined at compile time. It would be possible to extend the Transterpreter so that inter-processor channels could be created dynamically. A 'new channel' call could request that the PPU allocate memory for a new channel at runtime and inform the two processors that wish to communicate of the location in memory for this. An initial implementation would require a break from occam's normal semantics, much like the KRoC [SBW03] network channel implementation, to differentiate between interprocessor and local channels. It is possible to add explicit support for such channels to a compiler which would allow inter-processor channels to be transparent by automatically generating the required code for inter-processor communications.

One final limitation of the current implementation is that messages are limited to 16KB blocks. This boundary was chosen as it is the maximum data size that can be copied in a single DMA command. This, in itself, is not a difficult issue to overcome, the channel implementation can be extended to copy multiple blocks of data per channel communication as the DMA engine supports queueing of multiple data blocks. It has not been implemented until now as the need for larger messages has not arisen.

4.6 An evaluation of Mandelbrot algorithms for the Cell

In order to support the claim that occam-pi provides a good solution to problems of parallel decomposition and distribution, a Mandelbrot set generator using occam-pi on the Cell Transterpreter is presented, and the solution is compared to other programming models. The Mandelbrot set is a good example, because different sections of a Mandelbrot set require varying amounts of time to compute. This presents an interesting case for examining computational load balancing across the Cell. Two algorithms for distributing workload across SPUs are discussed and compared to similar solutions using OpenMP, MPI Microtask programming and CellSs. The general structure of the occam network as shown in Figure 12 does not change in the examples.

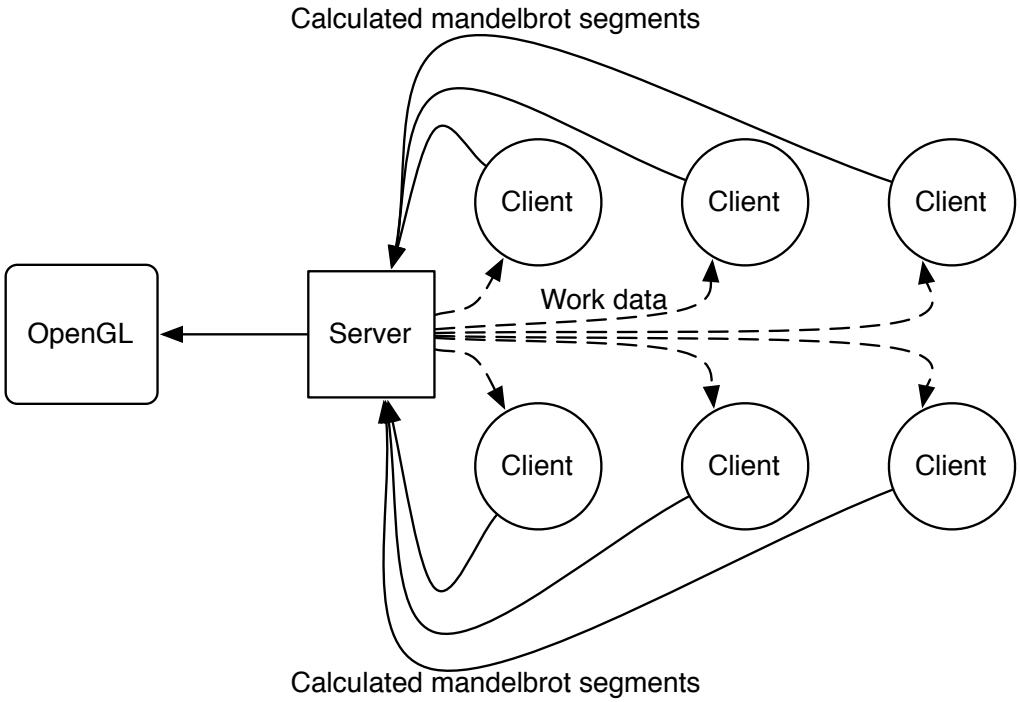

Figure 12: Parallel Mandelbrot on the Cell.

4.6.1 Overview

In the example occam-pi is used to provide the mechanism for distributing the work, while the computationally intensive portions of the program are written in C, for performance reasons. The Transterpreter provides all the communication facilities required

for the program and executes the occam portions of the program. The PPC runs a master process which handles workload distribution, the assembly of completed units and graphical rendering. Graphical rendering is accomplished using the SWIG generated bindings to SDL and OpenGL. The SPUs are assigned work by the PPC and compute sections of the Mandelbrot set for rendering. When computation of a work unit completes, it is sent back to the PPC and the SPU waits for the next work unit. Once an entire frame has been computed, the PPU draws the generated fractal to screen and assigns the work units for next frame.

4.6.2 A Simple Implementation

The first implementation mirrors an OpenMP-like behaviour, where the algorithm mimics the use of an `omp parallel for` directive. Work is distributed to all of the SPUs. Once all the SPUs complete computation, the results are collected and assembled into a frame that can be rendered. After the frame is rendered, the next parallel sequence of computation begins.

```
#define size 600
#define processors 6
#define stride size/processors
float frame[size][size];
while(true) {
  #pragma omp parallel for num_threads(processors)
  for (int x = 0; x < size, x += stride)
    calculate_mandelbrot(&frame, x, size);
  render_frame(frame);
}
```

Listing 4.13: OpenMP implementation of Mandelbrot pseudo code.

Listing 4.13 results in a division of the Mandelbrot into six slices of 100*600 pixels, each of which is computed on a single SPU. The rendering time for a single frame is bound to the thread with the largest amount of computation, leaving SPUs that complete their calculations faster idle. In order to mitigate this issue, it is possible to reduce

the work-unit size and allocate more processes to do the computation. If this is done, more threads are scheduled than there are SPUs, and an additional overhead of process context switch time is added. At a small enough granularity the overhead created through the cost of context switching SPU processes will outweigh any benefit provided through parallelisation. Such an implementation is inefficient as it requires that processes are continually started and stopped on the SPUs, further increasing overhead.

An algorithm implemented in occam-pi that tries to simulate the behaviour above will need, due to the languages and runtime design, a server process to farm and collect the work units, and client processes to perform the work. An advantage of such an approach is that it does not suffer from the start up and shut down time for SPU processes as the client processes run continuously. An algorithm mimicking OpenMP's behaviour is presented in Listing 4.14. The workload is first distributed to all the SPUs. Once work units complete, they are collated and rendered.

```
VAL INT PROCESSORS IS 6:
PROC svr([PROCESSORS]CHAN MOBILE []INT to.SPU, from.SPU)
  VAL INT IMAGEWIDTH IS 600:
  VAL INT stride IS IMAGEWIDTH/PROCESSORS:
  WHILE TRUE
    SEQ
      SEQ px = 0 FOR IMAGEWIDTH / stride STEP stride
        to.SPU[px/stride] ! [IMAGEWIDTH, px]
      SEQ
        SEQ i = 0 FOR PROCESSORS
          MOBILE []INT data:
          SEQ
            from.SPU[i] ? data
            copy.data.to.rendering.buffer (i, data)
        draw.graphics.buffer()
:
```

Listing 4.14: Simple server for a Mandelbrot set generator in occam-pi.

The client awaits work units and sends completed units back to the PPU. While awaiting new instructions from the server the end remains idle.

```
PROC client(CHAN MOBILE []INT from.ppu, to.ppu)
  MOBILE []INT data, work:
  VAL INT x IS 0:
  VAL INT y IS 1:
  WHILE TRUE
    SEQ
      from.ppu ? work
      data := MOBILE [work[x] * work[y]]INT
      calculate.mandel (work[x], work[y], data)
      to.ppu ! data
```

Listing 4.15: Client end of a simple Mandelbrot set generator in occam-pi.

4.6.3 Making use of explicit concurrency

A better implementation makes use of occam-pi's ability to explicitly control where data flows. This implementation supports the claim that occam-pi is a suitable language for developing concurrent programs and has a number of advantages over the pure C implementations of concurrency support such as OpenMP. occam semantics permit direct control over execution locality and provide safe mechanisms for inter-process and inter-processor communication, making the solution presented below difficult or inefficient to implement using a language with poorer concurrency support.

Listing 4.16 shows a better implementation of the server algorithm. It permits workers to receive new work units as soon as their current units have been completed. Each SPU has two worker processes; so that while one is sending a completed result, the other can commence work on the next set of data. By reducing the work units into smaller slices, processors do not have to wait for the entire set of units to complete prior to beginning the next set.

```
VAL INT PROCESSORS IS 6:
PROC svr([PROCESSORS]CHAN MOBILE []INT to.SPU, from.SPU)
  VAL INT IMAGEWIDTH IS 600:
  VAL INT PROCESSORS IS 6:
  VAL INT stride IS IMAGEWIDTH/PROCESSORS:
  WHILE TRUE
    PAR
      PAR py = 0 FOR IMAGEWIDTH / stride STEP stride
        SEQ px = 0 FOR IMAGEWIDTH / stride STEP stride
          to.SPU[py/stride] ! [px, py]
      SEQ
        PAR j = 0 FOR PROCESSORS
          SEQ i = 0 FOR PROCESSORS
            MOBILE []INT data:
            SEQ
              from.SPU[j+1] ? data
              copy.data.to.rendering.buffer (i, j, data)
        draw.graphics.buffer()
:
```

Listing 4.16: Server end of an efficient implementation of a Mandelbrot set generator in occam-pi.

The implementation shown in Listing 4.17 of the client runs four concurrent processes which allow computation and communication to overlap. The channels for communicating with the PPC are multiplexed on the SPU internally, and two work processes are assigned in a round robin manner. Thus computation and communication are allowed to overlap resulting in a better use of memory bandwidth, a technique which is known as latency hiding.

```
PROC client(CHAN MOBILE []INT from.ppu, to.ppu)
  VAL INT num.workers IS 2:
  [num.workers]CHAN MOBILE []INT workers, result:
  VAL INT x IS 0:
  VAL INT y IS 1:
  PAR
    WHILE TRUE -- send new units to workers
      INITIAL INT flip IS 0:
      MOBILE []INT work:
      SEQ
        from.ppu ? work
        workers[flip \ num.workers] ! work
        flip := flip PLUS 1
    WHILE TRUE -- collect and send complete units to ppu
      MOBILE []INT data:
      ALT i = 0 FOR num.workers
        result [i] ? data
          to.ppu ! data
    PAR i = 0 FOR num.workers -- start worker proc's
      MOBILE []INT data, work:
      WHILE TRUE
        SEQ
          workers[i] ? work
          data := MOBILE [work[x] * work[stride]]INT
          calculate.mandel (work[x], work[y], data)
          result [i] ! data
:
```

Listing 4.17: Client end of a Mandelbrot set generator with multiple workers.

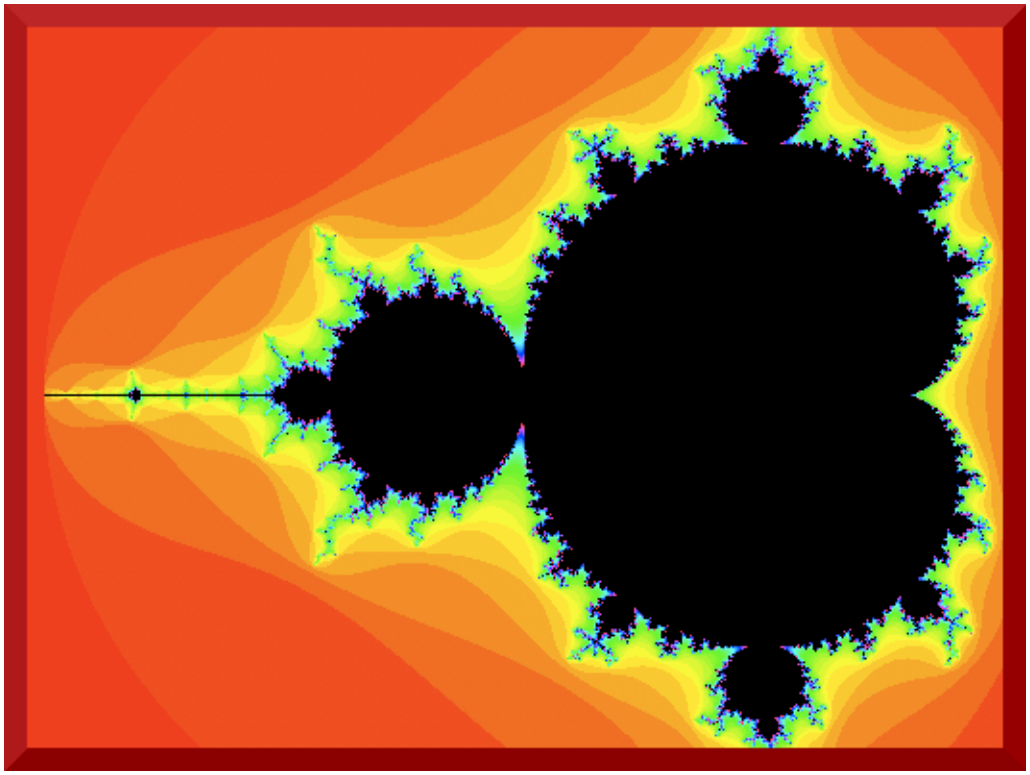

Figure 13: A Mandelbrot set rendered on the Cell.

4.6.4 Decoupling work units from SPUs

A third implementation makes another improvement to the Mandelbrot set generator. This implementation, as seen in Listing 4.18, modifies the server side to assign work units to all workers on start up. It then proceeds to use a single work queue to assign all new work packets as soon as processors become available. Work packets are now sent with annotations about their origin, and are assembled into a complete image once done.

```
VAL INT PROCESSORS IS 6:
PROC svr([PROCESSORS]CHAN MOBILE []INT to.SPU, from.SPU)
  CHAN MOBILE []INT sendon:
  VAL INT IMAGEWIDTH IS 600:
  VAL INT PROCESSORS IS 6:
  VAL INT stride IS IMAGEWIDTH/PROCESSORS:
  VAL INT workunits IS PROCESSORS * PROCESSORS:
```

```
  VAL INT x IS 0:
  VAL INT y IS 1:
  [workunits][2]INT worklist:
  SEQ
    -- precompute coordinates
    SEQ py = 0 FOR IMAGEWIDTH / stride STEP stride
      SEQ px = 0 FOR IMAGEWIDTH / stride STEP stride
        [worklist[py/stride] FROM 0 FOR 2] := [px, py]
    WHILE TRUE
      SEQ
        -- Send first work units
        PAR i = 0 FOR PROCESSORS
          to.SPU[i] ! worklist[i]
        INITIAL INT workposition IS PROCESSORS:
        SEQ j=0 FOR workunits
          MOBILE []INT data:
          ALT i = 0 FOR PROCESSORS
            from.SPU[i] ? data
              SEQ
                copy.data.to.buffer (data[x],
                       data[y],
                       data FROM 2 FOR SIZE(data)])
                to.SPU[i] ! worklist[workposition]
                workposition := workposition + 1
        --Reset and draw
        draw.graphics.buffer()
:
```

Listing 4.18: Server end of a farming implementation of a Mandelbrot set generator in occam-pi.

The client end, shown in Listing 4.19, is modified to take into account the additional coordinate data that needs to be passed around with the work units. This form of the

program should generally provide the best performance as none of the SPUs are likely to stall during computation. In the previous examples processors where liable to stall, should one processor's work unit take significantly longer than that of another. Because of the shared work queue, this is no longer an issue, as other processors can take on remaining work units regardless. The only place this example will stall is when a frame is completes and drawing to the screen. This could be improved by having the next frames work units sent out as the work units for the current frame complete.

```
PROC client(CHAN MOBILE []INT from.ppu, to.ppu)
  VAL INT num.workers IS 2:
  VAL INT x IS 0:
  VAL INT y IS 1:
  [num.workers]CHAN MOBILE []INT result, workers:
  PAR
    WHILE TRUE -- send new units to workers
      INITIAL INT flip IS 0:
      MOBILE []INT work:
      SEQ
        from.ppu ? work
        workers[flip \ num.workers] ! work
        flip := flip PLUS 1
    WHILE TRUE -- collect and send complete units to ppu
      MOBILE []INT data:
      ALT i = 0 FOR num.workers
        result [i] ? data
          to.ppu ! data
    PAR i = 0 FOR num.workers -- start worker proc's
      MOBILE []INT data, work:
      WHILE TRUE
        SEQ
          workers[i] ? work
          data := MOBILE [(work[x] * work[y])+2]INT
```

```
            --The first two values of data are
            --assigned to be work[x] and work [y]
            calculate.mandel (work[x], work[y], data)
            result [i] ! data
:
```

Listing 4.19: Client end of a farming implementation of a Mandelbrot set generator in occam-pi.

4.7 Further Work on the Cell Transterpreter

There are a number of further developments planned to the Transterpreter for supporting occam-pi programs on the Cell. Support for shared channels, barriers and other occam-pi features will be added. Further improvements will leverage the vector processing capabilities of the Cell BE architecture more effectively, and make better use of the underlying hardware for inter-processor communication. In the longer term, code generation and JIT mechanisms using the Transterpreter will be explored to gain significant performance improvements. Compiler support for creating arbitrary networks of inter-processor channels will be added resulting in more transparent and safer use of channels.

4.7.1 Arbitrary/dynamic channel networks

A method for abstracting the channel connections between processors, akin to the virtual channel implementation in PoNY [SBW03] would allow for multiple channels to exist between processors. This would enable a much more flexible mode of programming by multiplexing channels automatically.

4.7.2 Vector processing

The majority of the Cell's computational performance comes from the SPU processors. These are optimised to execute vector (SIMD) instructions. Adding support for SIMD,

therefore, would result in very large performance gains. Currently, it is only possible to make use of the SIMD capabilities of the Cell by writing C and calling it from within an occam program. Other options for making use of SIMD from occam are discussed in Chapter 5.

4.7.3 Code Generation

While being able to run occam-pi and the Transterpreter on a platform like the Cell is interesting [ocml07], it is not necessarily the optimal solution. Even though the runtime has a small memory footprint in comparison with other virtual machines, the limited amount of local store available on the SPUs becomes even smaller when it needs to be shared by byte code and the Transterpreter. Additionally, the current implementation of the Transterpreter replicates the same byte code to each processor, meaning that unused code is resident on each SPU, wasting further resources.

Furthermore, the overhead of running a byte code interpreter is particularly large since the SPU is only capable of 128-bit-sized loads and stores, and a penalty is paid on all scalar operations due to the masking and unmasking necessary for processing 32-bit values.

Due to the relatively high memory requirements, and the performance overheads of byte code interpretation, occam-pi and the Transterpreter can become unattractive to developers who need the specialised processing power that the Cell offers. In order to address these issues the idea of generating native code from occam-pi [JDC06] using parts of the Transterpreter runtime and gcc [GCC06a] is being explored. Such a solution will combine the speed of C with the safe concurrency that occam-pi has to offer.

A manner of either inferring automatically, or specifying through annotation, which parts of the code are required on which processor would allow for dead code elimination and hence, smaller binaries on the SPUs.

The current implementation of the Transterpreter for the Cell provides a basis for code generation for the Cell. It has shown that a language like occam-pi is usable on the Cell, and a different experiment has shown that efficient C can be generated from

occam-pi. Combining the knowledge of these two experiments should make it feasible to generate fast and safe programs for the Cell.

4.7.4 Efficient interpretation on the Cell

Running a switch-based virtual machine such as the Transterpreter on the Cell is not an optimal solution. Since SPUs pay high penalties for branching, a switched byte code interpreter is not the most suitable type of interpreter. In order to reduce the quantity of mandatory branches the Transterpreter could be modified to use direct-threaded interpreting [Bel73]. Such an approach reduces the number of branches per instructions to one, providing a significant performance boost on an SPU.

4.8 Summary

The long term goals aim to address some of the difficulties of programming the Cell processor that cannot be addressed exclusively through the use of preprocessor directives and loop level concurrency. Using the Transterpreter and occam-pi as a starting point, the language can be extended to provide direct support for modern hardware. occam-pi can be used to create efficient, parallel algorithms while providing the safety required for parallel programming without resorting exclusively to loop-level parallelism; this is illustrated in the Mandelbrot examples (Section 4.6). Client-server architectures can be built with relative ease using safe communication and synchronisation points. The prototype implementation of the Transterpreter for the Cell has shown that it is possible to use occam-pi on the Cell and it is a first step to bringing a more powerful programming model to this architecture.

Chapter 5

Vector Processing in occam-pi

Vector processing extensions that are commonly available on today's processors are particularly suited to most multimedia applications, and many scientific applications and simulations. Many of the desktop/server class processors that are supported by the CCSP and Transterpreter runtimes have such extensions, for example AltiVec on the PowerPC, SSE on IA32 based processors and particularly the Cell BE (which contains both an AltiVec unit as well as the SPUs which are effectively dedicated vector co-processors).

This chapter describes various ways of implementing support for vector processing using occam and the Transterpreter. The chapter begins by providing background and motivations for this work, followed by a discussion of previous work in languages based on the use of vector processing as well as how vector processing has been added to existing languages. A description of the implementation of vector processing support for the language and runtime is then presented, along with some of the challenges involved.

5.1 Vector processing background

Vector processing has been used for high performance computing since the 1970's around the time when the Cray-1 — a supercomputer that was capable of both high speed scalar and vector operations — was introduced. In the late 90's many desktop

class processors started implementing vector processing extensions to help accelerate multimedia applications [Int97]. At the same time graphics accelerators became mainstream and sported dedicated vector units which soon outperformed general purpose CPUs for vector processing tasks.

Despite the rapidly improving performance of GPU-based vector processing, many tasks are too large or too complicated to offload onto graphics processors, often requiring specialist programming skills or languages designed for vector processing such as Brook [BFH+04] or Cg [MGAK03] (C for graphics). The vector processing units on general purpose CPUs are currently still far more accessible to programmers and can provide good performance increases with a much smaller learning curve as well as being more generic and portable.

occam-pi natively has little support for data parallelism and does not provide explicit support for vector units. occam-pi's semantics, however, already enforce correct concurrent access to variables which would be amenable to the generation of vector instructions where appropriate, such as in the replicated **PAR** which can operate on sets of arrays. The combination of light-weight processes with vector processing built-in explicitly or transparently would permit simple programming for vector-enabled systems.

5.2 How vector processing is currently used

A number of different programming approaches exist which make use of vector processing either through explicit use of in-line assembly or intrinsics, or by language level support where vectorizable computations are specified through annotations or special syntax. In the latter case, much of the complexity of vector processing can be pushed to the compiler.

5.3 Extensions to GCC for vector processing

The GCC compiler [GCC06a], as of version 3, provides support for vector processing through intrinsic functions. These are designed to be as hardware independent as possible (although not compiler independent) and are relatively simple to use. Listing 5.1 shows how to instruct GCC 4 that vector instructions should be used for a given calculation. Since vectors are not a part of the language specification the code needed to make use of vector processing can be long-winded. GCC provides libraries for processors which lack either certain features or support for vector processing entirely, making the intrinsic vector functions more portable. In order for GCC to generate binaries that make use of vector extensions a command line flag must be provided, such as `-sse2` for Intel's SSE2 extensions or `-altivec` for a PowerPC's AltiVec unit.

```
/* Declare vector of four single precision floats */
typedef float __attribute__ ((vector_size(16))) vector4f;

/* Create a union type to ease
loading of data into the vectors */
typedef union vector4f_union {
  vector4f v;
  float f[4];
} vector4f_union;

int main(int **argc, *argv) {
  vector4f_union a, b, c;
  /* Load vectors with data */
  a.f[0] = 1.0; a.f[1] = 5.9; a.f[2] = 2.1; a.f[3] = 99.54;
  b.f[0] = 0.1; b.f[1] = 9.5; b.f[2] = 9.8; b.f[3] = 2.01;
  c.v = a.v * b.v;    //Multiply the vectors
  return 0;
}
```

Listing 5.1: Using GCC built-ins for vector processing

GCC also provides architecture specific `__builtin__` functions [GCC06b] which allow a programmer to make explicit use of hardware features available on a given processor. These functions are not portable and programs using these must be adapted for each new processor.

5.4 Stream programming

Moving away from intrinsics and built-in functions, a more concise approach to vector programming can be seen by looking at the Brook language [BFH+04]. BrookGPU is a compiler and runtime implementation of the Brook stream programming language for modern graphics hardware. It aims to simplify programming of GPUs for general purpose programming and it has a programming model that is amenable to them. Brook is a superset of the C language, where the primary addition to the language is a new data type called a Stream. Additionally, special functions are defined which operate on streams, called Kernels. A kernel function takes Streams as arguments and performs implicit for-loops on them. For example, Listing 5.2 shows a kernel that will apply the kernel `k` to all elements of the stream `s`.

```
kernel void k(float s<>, float3 f,
              float a[10][10],
              out float o<>) {...}
```

Listing 5.2: Brook vector programming example

Brook divides programs into small computational units and runs them through a vector processor. This ensures that long pipelines of data can be processed without any interruptions to computation caused by branching. Vector processors are kept busy and can come close to achieving their peak performance.

While the tool chain of the Brook language is designed for operating on GPUs, the model is interesting when applied to vector processing in general and is discussed further in the next section.

5.5 Vector processing in occam-pi

While language level support for vector processing was absent in the original occam, the notion that each operation can be viewed as a separate process has scope for making use of vector units. For example the replicated parallel **PAR** keyword allows a programmer to express very fine grained concurrency. Furthermore, the language allows for assignment-level parallelism, in one line of code multiple variables can be computed. This section explores how occam-pi can be used with vector processing, as well as describing new implementations for vector support in the language.

5.5.1 Using GCC vector intrinsics from occam-pi

The simplest way to add support for any hardware specific features in occam-pi is through its foreign function interface [WM99, DJ05]. A number of implementation specific issues need to be overcome for this to work. For example, the current occam-pi compiler is not able to align data to quad-words, which most common vector units require.

The most common vector instruction sets operate on quad-word sized vectors, often giving the programmer a choice of operating on either two 64-bit, four 32-bit, eight 16-bit or sixteen 8-bit values in parallel. In order for the vector extensions to be efficient, the data to be operated on must be pre-aligned to quad-word boundaries so that no time is wasted in fetching data from memory that is mis-aligned, significantly reducing potential performance gains.

In order to implement cross platform compatible support for vector processing, GCC's processor independent intrinsic vector functions are used. These functions, when wrapped, can be called from occam-pi to execute vector operations on vectors.

When calling these functions, data must be pre-loaded into a C data type that is correctly aligned for the vector processor to operate on it. This can be done by using an alignment aware version of `malloc()`, such as `posix_memalign()`. For example, on OSX, `malloc()` returns quad-word aligned values by default, but only returns word aligned memory on Linux, which is why the `posix_memalign()` call is required since

it allows memory alignment to be specified as a parameter. Listings 5.3 and 5.4 show the code required to create a set of vectors that can be used for vector processing from occam-pi.

Listing 5.3 begins by declaring the prototypes for the C functions that allocate memory for each of the vectors. It then defines the new **DATA TYPE** that stores a vectors memory address (pointer) its size. The example **PROC** takes two arrays of four single precision floating point values as parameters, and starts by declaring three `vec4f` variables that will store the pointers to the vectors. Finally, three vectors are allocated using the '`new`' functions.

```
#PRAGMA EXTERNAL "PROC C.vec4f.new.scalar (
                                  RESULT vec4f a,
                                  VAL [4]REAL32 b) = 0"
#PRAGMA EXTERNAL "PROC C.vec4f.new.empty (
                                  RESULT vec4f a) = 0"
DATA TYPE vec4f IS INT:
PROC example([4]REAL32 data.a, data.b)
  vec4f a, b, c:
  SEQ
    -- Load the vectors with numbers
    C.vec4f.new.scalar(a, data.a)
    C.vec4f.new.scalar(b, data.b)
    C.vec4f.new.empty(c)
    ...
:
```

Listing 5.3: Creating two vectors from scalar values and allocating a third, emtpy vector.

Listing 5.4 shows the C function that is called to allocate a new vector which copies existing scalar data in to it. The function that is called, defined on the the third line, is the first **PROC** prototype in the **#PRAGMA EXTERNAL** of Listing 5.3. As a parameter it gets a list of pointers which correspond to the arguments defined by the occam-pi prototype for that function. The function begins by declaring a counter as well as a

new variable of type `vec4f_union` which corresponds to the union type that is defined in the example shown for the generic `__builtin__` vector functions in Listing 5.1. Memory is then allocated for the vector, for brevity only the OSX `malloc()` version is shown. The pointer `newvector` is then assigned to the first argument of **PROC** `C.vec4f.new.vector`. Finally, the second argument passed to the function, which contains the values to be assigned, is copied into the newly created vector. The function corresponding to `C.vec4f.new.empty` only allocates an empty vector.

```
void _vec4f_new_vector(int w[])
{
  /*Allocate and assign pointer to vector*/
  int j;
  vec4f_union *newvector;
  newvector = malloc(sizeof(union vec4f_union));
  *(INT(w[0])) = (vec4f_union *)newvector;

  /*Copy vector data into vector*/
  float *res;
  res = (REAL32(w[1]));
  for(j = 0; j < 4; j++)
    newvector->i[j] = res[j];
}
```

Listing 5.4: Creating a vector in C.

Once the memory setup is completed, calculations can be made using native vector units. Listing 5.5 shows the call to the C function which will perform the operation as well as the functions prototype at the start of the listing. Listing 5.6 begins by assigning the pointer arguments passed to the function from occam-pi to C vector data types, and finishes by performing the actual operation.

```
#PRAGMA EXTERNAL "PROC C.vec4f.mul (RESULT vec4f c,
                                    VAL vec4f a, b) = 0"
PROC example([4]REAL32 data.a, data.b)
  vec4f a, b, c:
  SEQ
    ...
    C.vec4f.mul(c, a, b)
    ...
:
```

Listing 5.5: Multiplying vectors in occam.

```
void _vec4f_mul(int w[])
{
  vec4f_union *c, *a, *b;
  c = (vec4f_union*)(*INT(w[0]));
  a = (vec4f_union*)(VAL_INT(w[1]));
  b = (vec4f_union*)(VAL_INT(w[2]));
  c->v = a->v * b->v;
}
```

Listing 5.6: Vector multiplication helper function.

```
#PRAGMA EXTERNAL "PROC C.vec4f.free (vec4f a) = 0"
PROC example([4]REAL32 data.a, data.b)
  vec4f a, b, c:
  SEQ
    ...
    C.vec4f.free(a)
    C.vec4f.free(b)
    C.vec4f.free(c)
:
```

Listing 5.7: Freeing previously allocated vectors from occam

Once all the vector calculations complete, the memory previously allocated for the vectors must be freed. Listings 5.7 and 5.8 show how the memory previously allocated for the vectors is freed, in order to prevent memory leaks.

```
void _vec4f_free(int w[])
{
  free((vec4f_union*)(*INT(w[0])));
}
```

Listing 5.8: C helper functions for freeing occam vectors.

5.5.2 Performing more computations in C

While being able to make use of the vector hardware through C is useful, calling C functions, assigning pointer and freeing individual pointers creates a significant performance overhead. A more efficient solution is to allocate memory, copy data to vectors, operate on vectors and free memory in bulk. In order to do this the functions `C.vec4f.new.array` , `C.vec4f.array.mul` and `C.vec4f.array.free` can be used, and are shown in Listings 5.9, 5.10 and 5.11 respectively.

```
void _vec4f_new_array(int w[])
{
  int j, element;
  vec4f_union *newvector;
  int *unionpointers = (INT(w[0]));
  newvector = malloc(size);
  /* Create a list of pointers to the unions */
  for(element = 0; element < (VAL_INT(w[1])); element++)
    unionpointers[element] = (int)&newvector[element];

  float *res;
  res  = (REAL32(w[2]));
```

```
  /* Copy vector data into vector */
  for(element = 0; element < (VAL_INT(w[1])); element+=4)
    for(j = 0; j < 4; j++)
      newvector[element].i[j] = res[element + j];
}
```

Listing 5.9: A function that allocates an array of vectors and populates it with data.

```
void _vec4f_array_mul(int w[])
{
  int i;
  vec4f_union *c, *a, *b;
  /*Assign the arrays*/
  c = (vec4f_union*)(*INT(w[0]));
  a = (vec4f_union*)(*INT(w[2]));
  b = (vec4f_union*)(*INT(w[4]));
  /*Multiply all the arrays in a tight loop*/
  for(i=0; i< (VAL_INT(w[1])); i++) {
    c[i].v = a[i].v * b[i].v;
  }
}
```

Listing 5.10: A function that multiplies two lists of arrays and stores the result in a third array.

```
void _vec4f_array_free(int w[])
{
  free((vec4f_union*)(*INT(w[0])));
}
```

Listing 5.11: Frees an array of vectors.

These functions allow an arbitrary number of vectors to be allocated with a single function call, only requiring one call to `malloc()` for the entirety of the vector array. When operating on an array of vectors, the vector unit is kept busy operating at full

speed as its pipeline is kept full, with only the overhead of a single function call. Finally when freeing the memory allocated, only one `free()` call frees all of the memory previously allocated to an array of vectors.

While this is much more efficient, it results in a less consistent implementation. Having C functions that need to be called explicitly does not provide tight integration with the language, although it is better suited to the fast computations that the vector units on the processors provide.

5.5.3 Operator Overloading and Vectors

In order to provide a more consistent interface to the underlying C functions occam-pi's operator overloading is used. Operator overloading and its use for a variety of specialised floating point units on processors such as the SPARC or MIPS is described in Wood and Moore's [WM99]. When the KRoC tool chain was migrated to the Intel platform the support for specialised floating point units was dropped as it was platform specific.

Operator overloading in occam-pi allows a programmer to overload common operators such as '+', '−', '∗', '/' and other operators as shown in Tables 1 and 2. For example when operating on matrices, it would be useful to be able to denote whether one wants to find the cross product or the dot product in a matrix multiplication. Operator overloading is done by creating specially named **FUNCTION**s.

occam-pi **FUNCTION**s are not side-effecting. By allowing calls to the C language inside a **FUNCTION** it is possible to cause side-effects and extra care needs to be taken when doing so, in order not to break occam-pi semantics. The values passed to the function must not be modified in any way, and providing the C function calls are well behaved, the **FUNCTION**s semantics can be preserved.

The initial C implementation described in Section 5.5.1 can be used as part of the operator overloading mechanism. In order to overload an operator a special **FUNCTION** can be declared which tells the compiler how to operate on custom data types. Listing 5.5.3 shows how an operator can be overloaded for the new data type `vec4f` so that two

vectors can be divided while still making use of the processors vector extensions. The listing shows a **FUNCTION** with a special name that corresponds to the operator that is being overloaded. Two constants are passed as parameters, and a new vector variable is declared to contain the result of the **FUNCTION**. Memory is then allocated for the resulting vector, and the result is computed.

```
vec4f FUNCTION "/" (VAL vec4f x, y)
  vec4f result:
  VALOF
    SEQ
      C.vec4f.new.empty(result)
      C.vec4f.div(result, x, y)
    RESULT result
:
```

Using operator overloading a programmer can write `c := a / b` and still make use of a vector processing unit. Because the return value of the divide operation needs to be declared inside the **FUNCTION** the programmer must manually free the returned variables memory. The programmer may forget to deallocate the memory allocated for this new result, or overwrite the pointer to previously allocated memory, and thereby loose the reference for it, in either case the memory allocated is irretrievable.

Operator overloading in occam-pi has some limitations. It is not possible to create arbitrarily sized arrays in functions, since memory allocation is side-effecting. The implications are that one cannot make use of the performance increase that operating on arrays of vectors simultaneously brings through operator overloading using this technique.

A possible solution to this problem would be to have a **FUNCTION** which takes two arrays of vectors as arguments, and returns a pointer to an array of vectors which could then be copied into an actual array of vectors. For example Listing 5.12 defines a new **DATA TYPE** which is a structure consisting of two elements, namely, a pointer and the pointers size.

```
DATA TYPE vec4f.array
  RECORD
    INT ptr:
    INT size:
:

vec4f.array FUNCTION "/" (VAL []vec4f x, y)
  vec4f.array result:
  VALOF
    SEQ
      result[size] := (SIZE x)
      C.vec4f.new.array.empty(result[ptr], result[size])
      C.vec4f.array.div(result, x, y)
    RESULT result
:
```

Listing 5.12: User defined operator for arrays of vectors.

Inside the user defined operator **FUNCTION** "/" memory is allocated for the result according to the length of the parameter arrays. Once allocated, the result of the division of the `x` and `y` vector arrays can be computed and stored in the `result` variable. Should the programmer then wish to manipulate individual elements of the array, a new supporting function which converts a `vec4f.array` to a `[]vec4f` can be used. It would be also be possible to only use `vec4f.array` data types for arrays of vectors, although this would make manipulating individual vectors long winded.

5.6 Vector support for current runtimes

This section explores how vector support can be integrated into the existing occam runtimes directly, providing the programmer with better support for vector processing.

5.6.1 Automatic vectorisation in occam

Ideally, it should be possible to vectorise scalar code automatically that is inside a replicated **PAR**allel loop. Listing 5.13 shows an example of such a loop. **PROC** `example` takes three arguments and performs the same calculation on each of the elements in parallel. In the simplest cases, the compiler already knows that it is safe to perform this operation in parallel, since it has already checked the operation is valid. Translating this to vector code would require the compiler to allocate correctly aligned memory, and then generate the corresponding vector instructions.

```
PROC example([]REAL32 a, b, result)
  PAR i = 0 FOR (SIZE a)
    result[i] := a[i] * b[i]
:
```

Listing 5.13: A **PAR** that could benefit from automated vectorisation.

This idea is somewhat similar to the annotation that is used in OpenMP, with the advantage that occam-pi does not permit aliasing and the compiler is able to perform compile time checks for parallel usage correctness inside a **PAR** block. No further annotation is required in this case in order to infer that such a **PAR** is safe to vectorise. By aligning all variables to quad-word offsets the compiler could be written so that it always generates vector instructions for the above.

An vector specific extension to occam-pi based on these ideas is proposed and can be implemented to support either explicit or implicit vector operations [Dim06]. For example a tool would be created that automatically vectorises ETC code and generates a new, vector extended ETC equivalent. These new vector instructions could be added to the Transterpreter to prototype such a system, and eventually also added to the tranx86 tool. Alternatively, explicit support for using vector operations could be added to current occam-pi compilers to permit finer grained control over when vector operations should or should not be used.

5.7 Benchmarks

In order to evaluate the utility of the developed vector libraries and to indicate how a full implementation of vector support would perform in occam-pi the vector implementations that were discussed in this Chapter were benchmarked. The values given in the last column of Table 5 are relative values of the total time, the cost of initialisation of a large array of vectors, the time the actual calculation took and the time taken to free the memory after the calculation.

All of the tests perform the same calculation, two vectors are multiplied, and then divided one hundred thousand (100,000) times in each test. The operation shown in Listing 5.14 illustrates the type of calculation performed. All tests where performed on the Transterpreter running on a 3.4 Ghz Pentium 4 with SSE3 instructions.

```
-- loop.len is 100,000
-- vec.len is 4  (4 * 32-bit values,
-- which fit into the 128-bit wide SSE)
SEQ i = 0 FOR loop.len
  SEQ j = 0 FOR vec.len
    SEQ
      c[i][j] := a[i][j] * b[i][j]
      c[i][j] := b[i][j] / a[i][j]
```

Listing 5.14: Vector calculation benchmark pseudo code.

Test nr 1 is a pure occam-pi implementation of the problem and is used as the reference point for our calculations. Initialisation and freeing are free as all data is statically allocated on start up. Tests 2 and 3 both perform all calculations in C, and only differ in how the data is initialised. In test 2, the data is loaded into C vectors by passing four values from occam-pi to C as shown in Listing 5.15.

```
C.vec4f.new.scalar(a[i], 1.0, 5.0, 10.0, 15.0)
```

Listing 5.15: Allocate a vector in C from four values.

Test 3 assumes pre-loaded arrays in occam-pi (such as when data may have been loaded from a file), and passes a pointer to each occam-pi vector as shown in Listing

5.16.

```
-- "avector" is defined at an earlier stage in
-- the program, ie. by loading data from a file.
C.vec4f.new.scalar(a[i], avector)
```

Listing 5.16: Allocate a vector in C from an occam array.

The tests 2 through 4 make use of vector processing units while tests 5 and 6 only make use of scalar operations in C and occam are are included for reference.

Test 4 shows the time taken for initialisation and freeing when it is done in large blocks. The calculations in this test are also done in bulk, resulting in only two calls to C in order to multiply and then divide the entire vector array. This very large relative speed up shows the cost of repeated foreign function calls, multiple calls to malloc and free, as well as the overhead of interpreting byte code.

Test 5 shows the amount of time it takes to compute the data when operator overloading is used. The cost of pre-loading the occam-pi arrays of data that are copied into the vector memory is not included to show only the cost of a single malloc(). While this method is probably the easiest to grasp, it is also the slowest.

The 6'th test performs the same operations simply using occam-pi arrays of four floating point numbers which are passed to C and are computed on the normal floating point unit. It has lower initialisation overhead as no calls to malloc need to be made and no additional cost is paid for freeing the memory as this is done automatically. It is also faster than methods 2, 3 and 5 as the cost of many additional C calls outweighs the benefits of using a vector unit.

5.8 Conclusions and future work

The results from the experiment above are encouraging as they indicate that there is a performance gain to be had from using vector operations from occam-pi. A full implementation would permit a large speed up for vector operations when used in conjunction with the Transterpreter or tranx86 and CCSP.

Test nr	Calculation Type	Total in ms	Init/alloc in ms	Calculation in ms	Free in ms	Relative Speedup
1	scalar occam	777,885	0	777,885	0	1.00x
2	vector	483,411	216,893	120,544	145,974	1.61x
3	vector array alloc	449,146	183,930	119,782	145,434	1.73x
4	vector pipelined	23,497	7,621	14,842	1,034	31.11x
5	vector overloaded	583,736	167,228	267,571	148,937	1.33x
6	scalar C	191,289	0	191,288	1	4.07x

Table 5: Sample results for vector processing in occam-pi.

Currently Quad-word alignment is being added to the OCC21 compiler and could be added to Tock. New vector specific instructions will extend the ETC definition and implemented in the Transterpreter and tranx86. Automated vectorisation support could then be added to an occam compiler which would make use of these instructions. It may also be possible to force quad alignment of all **MOBILE** data on the Transterpreter by modifying the instructions for allocating and de-allocating data.

Tock would be instrumented to use aligned memory/vectors for variables or arrays, and then automatically generate GCC intrinsics where appropriate. This would make vector support much simpler, although at the cost of a slight increase in memory footprint if an application uses many of simple variables as opposed to arrays. A less naive implementation could also infer which variables require alignment and align these as appropriate.

Chapter 6

Further contributions

This chapter describes some of the further contributions made by the author and is divided into four sections. It begins with a discussion of how occam-pi is used in the context of the Player/Stage library for robotics education. The chapter continues with a brief discussion about the prototype occam compiler '42' that was co-developed by Jadud and the author. Finally, a summary of the additions and extensions that where made to the Transterpreter by the author is presented, and some of the ports of the Transterpreter that been investigated.

6.1 occam-pi and Player/Stage

Player/Stage is a robotics simulation and programming environment [BCG+05]. Player is the robotics programming API part of of Player/Stage, and can be used for controlling virtual robots inside Stage, a 2D robotics simulator. Player can also be used as a standardised API for the robotic platforms that are modelled by Stage, enabling a programmer to develop programs for a simulated robot inside Stage and then to run the same program on an actual robot.

Figure 14 shows a model of a Pioneer III robot in a simulated environment created using Gazebo, Stage's 3D simulation equivalent. Stage and Gazebo have an accurate physical model that is used to simulate user-created environments in which one or more robots can interact.

occam-pi's parallel constructs can be mapped to a robots actions in the real world. Channels can be used to describe sensor input and motor output, and processes can be used to describe concurrent actions that occur in an active, real world environment. Jacobsen and Jadud also propose that robots provide an intrinsically motivating environment for teaching concurrency using occam-pi [JJ05]. On this basis, an interface to the Player API was created using the tool described in Chapter 3, permitting the control of both real and simulated robots.

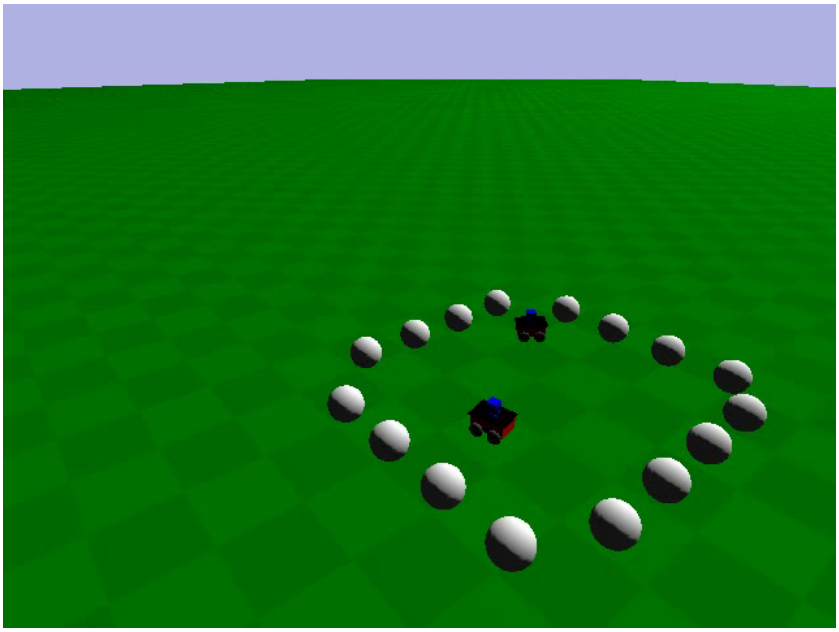

Figure 14: Two Pioneer robots in the Gazeebo virtual environment controlled from occam-pi.

This interface is used to create a channel and process oriented robotics abstraction in occam-pi for teaching concurrency and forms the basis of RoboDeb [DJJS, JJ07], a complete robotics simulation and programming environment for teaching concurrency and parallelism. RoboDeb provides a complete IDE for the occam-pi programming language and leverages the Transterpreter along with the Player/Stage to provide robotics simulation. RoboDeb is built on the basis of the Debian operating system and is designed to run inside VMWare's Player [VMw] (a free virtual machine environment) so it is be easily downloadable and runnable by students.

RoboDeb has been used in a number of courses such as Extreme Multiprogramming [Vin07] at the University of Copenhagen, CO631 Concurrency Design and Practice [Wel09] at Kent University, and more recently in Programming in Parallel Pedagogically [JSJ08] at Olin College. The Player/Stage bindings have also been used by Simpson [SJJ06], where the Subsumption architecture is explored in the context of occam-pi and robotics on a Pioneer III robot.

6.2 The 42 compiler

During the course of this research it became evident that in order to achieve some of the future goals this thesis presents, it is necessary write a new compiler. The primary motivations for writing a new compiler were:

- Easy to read - each individual transformation of the syntax tree should be contained in one pass.
- Easy to extend and experiment with new language features - language features should be easy to add, new features or modifications should be as non-intrusive as possible, mainly requiring the addition of new passes, and few modifications to the existent compiler. Many of the Occam Enhancement Proposals (OEP's) [Sam06] are non-trivial to implement in the current compiler.
- Error messages provided should be as helpful as possible, with customised error messages for as many cases as possible so that the language(s) it supports are better suited to educational contexts.
- The parser should be simple and use a rich parsing framework, so that modifying or experimenting with a new syntax for existent features such as the vector extensions proposed in Chapter 5 will not be overly time consuming.
- The compiler should be able to support multiple front-ends and, where possible, multiple variants of occam.

- The compiler should support multiple back-ends, including transputer bytecode output and C generation, not too dissimilar to the SPOC compiler [DHWN94].

OCC21 is not considered a good choice as a starting point for our goals as its original design goals were to be a compiler for a purely static occam, and it was not designed to be extensible. OCC21 is written in C and designed to run on memory constrained systems. As a result parts of the code are difficult to read and maintain. C is also not the most suited language for writing compilers, which predominantly need to recurse over large lists (parse trees). It lacks good support for string manipulation, it lacks modern parsing tools and it is difficult to implement algorithms which check for parallel correctness. In fact OCC21 had, and to a large degree still has many corner cases where it fails to detect parallel usage of variables or channels correctly. This can prevent the compilation of correct programs; In addition it occasionally results in cryptic warnings or errors. C's lack of good memory management tools also results in the compiler occasionally suffering from segmentation faults during compilation. While many of these problems have been corrected, OCC21 is still difficult to work with. It currently consists of around 113,000 lines of C. Adding a new feature usually requires the compiler to be modified in multiple places, and required an extensive understanding of its internals to work with reliably. These factors also contributed to the desire to write a new, cleaner, simpler to maintain compiler in a higher level language with better parsers.

In order to study the feasibility of a new compiler, an experimental compiler named '42', the precursor to the Tock [SB08b, SB08a] compiler, was written in MzScheme. MzScheme a concise functional language [Fla01] that made it easier to construct a nano-pass compiler [SWD05]. MzScheme's rich macro system and functional nature, allowed a syntax to be created specifically for parsing, modifying and compiling an occam-like syntax into transputer byte code. MzScheme has excellent support for recursion, list and string manipulation, rich pattern matching. These high level features make it a better choice of language for a compiler than the commonly used C. The choice of MzScheme was enforced by good knowledge of the language in the development group, because the Transterpreters linker was already using it, meaning many of

the people who would be involved with the project would already have some familiarity with the language.

The '42' compiler is a nano-pass compiler and only performs one type of transformation or check on the syntax tree in each pass, where possible. The nano-pass architecture allows each transformation or check to be enabled or disabled during compilation. New or experimental passes can be inserted at any point in the compilation process allowing for new language features to be prototyped or added more rapidly than before. '42' is able to compile a subset of occam-like syntax to transputer byte code and forms the basis for the work on occam-pi to C generation [JDC06], where the use of C as a portable high level assembly for occam-pi is evaluated.

'42' is able to parse a subset of occam based syntax in the form of S-expressions, as shown in Listing 6.1. It supports the **SEQ** and **PAR** constructs, as well as **PROC**ess declaration, **CHAN**nels and the type **INT**etger. It is able to compile basic programs to C and transputer bytecode, the latter can then be executed on the Transterpreter.

```
(module declare-and-add soccam
  (proc main ()
    (decl ([a int]
           [b int]
           [c int])
      (seq
       (:= a 3)
       (:= b 5)
       (:= c (+ (- a a) b))))))
```

Listing 6.1: A sample S-expression occam program.

'42' has now been superseded by a new compiler written in Haskell named Tock, whose design and implementation are inspired by '42'. Tock started out as a front-end parser for '42' to translate occam-pi to the S-expression based syntax, and later turned into a fully standalone compiler of its own, overtaking '42' in its functionality. Tock is set to become the new occam-pi compiler and may replace OCC21 as it has more robust test coverage and is able to provide wider coverage for detecting concurrency

induced error conditions that the current compiler cannot. Tock also supports a new concurrent language Rain [Bro06, BW03b], building on ideas from occam-pi and C++. Furthermore, Tock is able to generate CIF code for use with CCSP, or C++ for use with the C++CSP [Bro07] library. Tock fulfils most of the goals originally set out in '42', barring transputer byte code output. Support for transputer byte code, and the Transterpreter can be added in future.

'42' was developed by Jadud and the Author. The C back-end was co-authored by Jadud and Jacobsen, and described in more detail in Jacobsens [Jac08]. Tock was developed by Brown and Sampson.

6.3 Extensions to the Transterpreter

A number of extensions to the Transterpreter runtime have been made by the author during the course of this thesis. The Transterpreter originally supported integer hardware, and lacked native support for barriers and floating point operations in occam-pi as these required additional instructions to be present. The addition of barriers and floating point operations to the Transterpreter also provides an accessible reference implementation.

Floating point support was initially available on the Transterpreter only in the form of floating point libraries written in occam that simulated floating point operations on integer hardware. While this is useful on hardware without native floating point support, it does not take advantage of floating point hardware when available. Floating point support was added to the Transterpreter by the author by studying the "Compiler Writer's Guide" [INM88], and incorporating changes to the Transputers original floating point instruction set introduced in the newer transputer byte code (TCE) format. In order to implement the floating point instructions three new virtual registers, the floating point registers, where added to the Transterpreter. The floating point instructions rely on the new registers for floating point operations.

Support for barriers was originally also only available through support libraries, and not directly supported by the runtime. This however was insufficient, as a newer syntax for barriers was added to occam-pi which required supporting functionality in

the runtime. The original KRoC implementation, written in IA32 assembly, was used to implement aa new, more accessible and portable, version in C. This was then built into the Transterpreter, giving it full support for occam-pi's barrier syntax.

One of the primary side-effects of the attempted robotics ports was that the Transterpreter had to become more modular, it is now possible to include or exclude portions of the available instructions depending on platform requirements. Further improvements to the Transterpreter include: a number of 16-bit specific bugs that where ironed out, 64-bit floating point calculation support on big-endian platforms along with hardware floating point support in general.

The author also added support for the conditional compilation of various features of the Transterpreter runtime. This allowed it to run a subset of the language on smaller, more memory constrained hardware which did not require all of occam-pi's features, such as barriers, mobility or hardware floating point support, and can save a few kilobytes of space.

Initially the Transterpreter also had a limited C interface which required C libraries to be statically compiled in with the runtime. The author modified the Transterpreter so it could load dynamic libraries, making the C interface more usable in general. The dynamic library loader was designed to work on any platform that supported dynamic libraries and the Transterpreter.

Finally the author ensured correct behaviour of the Transterpreter on big and little-endian machines. As the Transterpreter byte code is little-endian, care needs to be taken when running the Transterpreter on big-endian machines such as the Cell or the Lego Mindstorms. Many instructions had initially assumed both little-endian data and byte-code, or in the case of 64-bit values, ordering, which resulted in instability or incorrect behaviour on big-endian machines. The Transterpreter now runs correctly on both little and big-endian machines, and passes all of the CGTests — the occam-pi compiler test suite [INM10].

6.4 occam-pi on other platforms

While exploring viable platforms for robotics control hardware the Transterpreter was ported to a number of new platforms by the author. Some of these ports are now of limited utility due to hardware obsolescence, or other practical reasons such as inadequately supported compilers, they were, however, useful as tools for the further development and debugging of the Transterpreter.

6.4.1 16-bit DOS

This is a port of the Transterpreter capable of running in 16-bit mode on DOS. The purpose of this port was two fold. It was created to allow people to test programs written for the LEGO Mindstorms on a 16-bit platform that had text-output and debuggers and second it allowed the CGTests test-suite on a 16-bit platform to help find any problems related to the incorrect implementation of instructions in the virtual machine in 16-bit mode. The 16-bit Transterpreter for DOS was compiled using the Open Watcom compiler [Ope08].

6.4.2 Symbian S60

A platform that was evaluated in the context of robotics (for use as the controlling portion of a robot) was the "S60" [Nok09] series of mobile phones produced by Nokia. These widely available devices contain a phone with a GPRS unit, a digital camera, blue tooth, USB and run the Symbian operating system (version 6 or newer) on an ARM9 processor. Mobile phones tend to have good battery life and low power requirements relative to other portable computing devices, coupled with network connectivity facilities they are an interesting platform for robotic control.

An S60 based phone also has blue tooth which would allow a robot to connect to a GPS for navigation. The robot's servos could be controlled via the USB port using a controller board with connections for actuators and sensors. The built-in camera could be used for visual feedback.

The freely available GCC-based tool chain provided by Nokia makes it relatively easy to cross-compile for Symbian based S60 mobile phones. A version of the Transterpreter which connected with Symbian's C++ API was written, and simple occam programs could be run on S60 based mobile phones using text input and output. The presence of a relatively feature-rich operating system simplified the process of porting the Transterpreter.

Unfortunately the Symbian operating system is written entirely in C++ and no C APIs are available. The lack of support for generating bindings to C++ libraries in the occam module for SWIG made the creation of good bindings to Symbian libraries for any of the hardware prohibitively expensive in time. The port, while feature-complete, had some issues with larger occam programs as the loading of the byte code would fail when allocating memory over a few kilobytes, which caused the Transterpreter to hang. Despite this, a percentage of the CGTests tests passed successfully on the Symbian phone.

While this platform has potential uses as a robotics controller for occam programs, the lack of good support for C++ libraries, mean that the implementation could not be completed within a reasonable time-frame. S60 based devices remain interesting platforms to pursue from a robotics perspective and, once C++ support is added to SWIG for occam-pi, development of this platform could resume, as interfacing with Symbian's libraries will be greatly simplified.

6.4.3 Gameboy Advance

The Gameboy Advance was evaluated as a potential robotics platform as it has a number of interesting features, is small and contains a powerful processor. The Gameboy's hardware includes wireless communication, a screen and a USB connector as well as good battery life. It is powered by an ARM9 based processor and has sizeable memory. It is also possible to get relatively inexpensive memory card interfaces for it that enable the creation of user software, without requiring an expensive development kit or a licence from Nintendo. At the time of the evaluation it was relatively easy to compile

a version of GCC which could cross compile programs for the Gameboy Advances architecture. A few emulators are also available making it possible to develop programs without requiring hardware.

A port of the Transterpreter to the hardware was begun, but due to the lack of good documentation about the system it was soon abandoned. One of the primary difficulties was getting the simulator to display information on the screen as the open source Gameboy tool chain did not have a standard library for text/video output. It was also not possible to use tools such as GDB for debugging programs, as the freely available simulators where geared towards playing existing games and not developing software. Finally, time constrains rendered the porting attempt to be infeasible. In future the Gameboy Advance, or its successors, may be a more compelling platform for robotics as the hardware will not only be more affordable, but it may also have substantially more documentation that is freely available.

6.5 Summary

The additional contributions presented in this chapter provide a basis for future work. While the individual contributions are mostly small experiments, they provide a basis on which further developments could be made. As a prime example of this, the Tock project, now well developed, evolved from the ‘42’ experiment. The following chapter presents conclusions and future work in greater detail.

Chapter 7

Conclusion

This thesis demonstrates that the occam-pi programming language, combined with the Transterpreter, provides a set of abstractions that are well suited to modern parallel processors and provides a path for the parallelisation of existing programs. In order to demonstrate this claim it addresses three key areas in the language's tools and syntax that are likely to impact on the practicalities and further growth of the occam language. The three areas explored are: interfacing with existing libraries, extending runtime support for new architectures; and providing compiler and semantic support for hardware features.

The first area provides an automated, reusable mechanism for interfacing with existing libraries and software has led to the development of new research and educational tools that would have not been possible previously. Virtually all new occam-pi applications benefit directly or indirectly from the creation of this tool.

The second area resulted in the adaptation of the Transterpreter runtime for the Cell. This resulted in new ways of using modern hardware to support occam-pi channel communications. It has also resulted in a new platform being available for exploring parallel algorithms implemented in occam-pi. Furthermore, the Transterpreter was also evaluated on a number of different platforms, and these experiments have been documented.

Finally, language extensions that can make occam-pi a stronger choice for software development have been identified, and a prototype compiler was developed, whose

ideas form the basis of a new, fully featured occam-pi compiler.

The Transterpreter has proven to be a good platform for developing and testing new paradigms and porting occam-pi to new hardware.

7.1 Contributions

The contributions that this thesis makes are presented here:

- SWIG occam interface - Automated support for generating language bindings has made programming in occam-pi more accessible and has helped in the implementation of many new research projects, and is used in virtually all new/recent occam applications.
- The Cell Transterpreter experiment shows that a language like occam-pi is suitable for the Cell, and explores techniques for channel communications.
- A comparison of workload distribution on the Cell, comparing the OpenMP, MPI Microtask approaches to occam-pi and CSP-related approaches, using a Mandelbrot fractal simulation as an example.
- An evaluation of possible avenues for implementing vector processing support in occam-pi, including a proposed specification of how vector processing could be implemented.
- Extension of the Transterpreter runtime to support native floating point operations and barriers, as well as providing accessible reference implementations for instructions to support these. Testing was conducted to ensure that the Transterpreter executes correctly on big and little endian platforms and many endian-related issues were corrected. More tests where conducted to ensure correct operation on both big and little endian 16-bit platforms.
- Construction of a prototype compiler for occam-pi, proving that it is possible to build an easy-to-extend and use compiler for the language thereby helping to

define requirements that a new fully featured compiler would need. The work presented in '42' has directly influenced the creation of a new, safer and more extensible compiler for occam-pi.

7.2 Short-term future work

This section presents a number of shorter term plans for future work, which aim to address more immediate needs for the further development of the occam-pi language and runtime.

7.2.1 Automatically generating CIF for Callback functions

Adding support for the occam-pi C-interface (CIF) could help extend SWIG interfaces. Since C code can be embedded into SWIG interface files it is desirable to extend SWIG to support annotations for callback functions, global variables and shared data to provide automatically generated and safe(er) interfaces to these. For example, libraries which make use of callbacks can have customised **PROC**s written in CIF which would be run on program start-up. **PROC**(s) would have a channel interface which can notify an occam-pi **PROC** of callback events such as keyboard and mouse presses. Exploring the possibility of embedding CIF into SWIG wrappers is also part of the future work planned.

7.2.2 Support for C++ from occam-pi

The addition of support for C++ will make occam-pi more accessible for platforms based on the Symbian operating system such as the Nokia S60 series of telephones. Symbian is written in C++ and, in order to write meaningful programs for it, interfacing with Symbian's C++ libraries is required. Once C++ support is added to SWIG's occam-pi module, it will be possible to resume exploration of the S60 telephone platform as a robotics controller using occam-pi.

7.2.3 Standardised graphical interface

As occam-pi lacks a standardised graphical interface, a logical enhancement work would be to try to establish a standard layer for graphical user interface (GUI), that is part of the languages operating environment. This could be achieved by including a set of top level channels which deal with graphical input, and keyboard and mouse input. Such a framework could be based on the work presented in WGOccam and GTKOccam [Sto03, Whi00], possibly also integrating idea present in Sampson's "occade" games programming environment. A new graphical manager could be devised to enable programming powerful graphical models in occam-pi. This would make the language more suited to both desktop application programming as well as aid in the visualisation of scientific simulations written in occam-pi.

7.2.4 Transterpreter Cell enhancements

A number of Cell specific enhancements are planned to the Transterpreter. An outline of these is given below.

Full interprocessor occam-pi support

While channel communication is now supported by the Transterpreter on the Cell BE, a number of other features of the occam-pi language still need to be added. Notably, support for barriers and shared channels across multiple processors as well as distributed deadlock detection are currently lacking. As algorithms to support these features already exist in CCSP for multi-processor, shared memory computers, the existent algorithms should be adaptable to the Cell BE with some thought.

Dynamic code loading

The existing operating environment could be extended to support dynamic loading of code on the Cell. Such dynamic code loading is already in use on embedded platforms where uploading new byte code is very slow. The embedded platforms only need to have the Transterpreter and its operating environment pre-loaded once, allowing new

user space programs to be uploaded without having to re-program the entire machine. For an SPU the benefit would be that user programs could be swapped out with a memory-resident Transterpreter, requiring much smaller amounts of program data to be swapped, overcoming much of the overhead that process swapping currently has on SPU's. The operating environment could also be extended to support dynamic creation of inter-processor channels at runtime, making user programs more flexible by making it possible to have arbitrary channel configurations between processors.

Replace top level process signature to transparently have interprocessor channels

Newer revisions of the Transterpreter make it easier to modify the top level process signature. Updating the Cell Transterpreter to use the newer code base will permit integration of the current occam support environment, currently provided as a library to be integrated into the Transterpreter. The process of adding support for this is already under way.

Evaluation of more efficient channel communication methods.

The current channel implementation on the Cell has scope for improvement. Currently, during channel communication system memory needs to be polled if a channel is awaiting input, in order to determine whether a channel is ready for reading. An alternative means could be explored, by for example using the PPC to marshall channel communications. One option would be to have the PPC poll the channel words to see if any channel communications have completed, and notify the reader SPUs through the mailboxes that they have data waiting for them. This may prove to be more efficient as it would not involve as many DMA calls to main memory, and may require slightly less memory bandwidth. Another option would be to evaluate the use the interrupting mailboxes as a means of notifying the PPC that a message is ready for another SPU.

A key factor for performance and usability on the Cell BE will be support for memory alignment in **MOBILE** data. Once **MOBILE** data is aligned correctly for the Cell, DMA transfers can be made much more efficient by removing the need for an additional copy to aligned data, as well as reducing the requirement that twice the memory

of the packet being sent is needed for each communication.

An optimisation for small data packets could be added to the current channel communication method. Since the channel word only requires 32-bits of memory, and all DMA requests have a minimum size of 128-bits, it would be easy to pack channel communications that are shorter than three 32-bit words into the same DMA that sets the channel word. This would reduce the number of DMA requests by two, one on the writing and one on the reading side, as only one DMA would be required from each side.

7.2.5 General Transterpreter enhancements

A number of enhancements to the Transterpreter have been identified which could improve it's performance and functionality both on and off the Cell.

Support for Explicit or Implicit memory alignment

The already mentioned support for specifying memory alignment for `MOBILE` data would not only help with channel communication on the Cell. Support for aligning memory can also be used to implement more efficient support for vector processing, since `MOBILE` data that has already been aligned can be operated on directly by a processors vector units, without requiring that it be copied to correctly aligned memory. Memory alignment is already supported through annotation of the code in OCC21 and CCSP.

Direct Threading

Currently two versions of the Transterpreter exist — a switch based mechanism, and a jump-table based mechanism. The switch-based version, which consists of a very large `switch` statement that is used to decode each instruction, is generally faster than the alternative because switch statements are compiled efficiently by GCC. It also results in smaller binaries, which makes it a more attractive option for embedded targets.

The jump-table mechanism is an array of function pointers which decodes instructions by using instructions as indices into the array. The jump-table mechanism was slightly easier to implement to start with, and was able to provide more detailed debug information at runtime. For example, indices which did not have functions allocated to them would have functions that notified the user that they where not implemented, or invalid, depending on the instruction number. This helped in the development of the interpreter, allowing the developers to determine which instructions where required for the correct operation of the virtual machine. An added side-effect of using an array of function pointers meant that, in theory it should be possible to enable, disable or even replace instructions at runtime. For example, a user program could request that all channel communication instructions be changed to become networked channel communication instructions, and have the Transterpreter spread itself onto a network via some mechanism such as MPI. While jump-tables have advantages to debugging instructions, and possibly experimenting with new features, they are quite slow as the additional memory lookups result in worse performance.

A different approach to interpretation, however, supplanting both the switch and jump-table mechanisms would be to use a direct threaded interpreter. This approach is advocated by an experimental JVM for the Cell BE [WNGG, NF08]. Direct threading turns all instructions into function pointers so that each new instruction can be called directly. This reduces the number of branch instructions by one each time an instruction is executed. Given the high cost of branching on an SPU, this could provide a significant performance boost for the Transterpreter, both on and off the Cell BE. There is a cost associated with direct threading however. Currently transputer byte code is Huffman encoded, and uses either 8-bits for primary instructions or 16-bits for secondary instructions. Converting each instruction into a function pointer would mean that the instruction stream would require up to four times as much space in memory. In order to implement direct-threading, all the jumps in the code would need to be updated to the new longer offsets. The already existent jump-tables could be used to transpose an existing instruction stream to a direct-threaded stream, so some of the work required to achieve this is already in place.

7.2.6 A more exhaustive comparison of algorithms

Further research is needed into the implementation of common algorithms on the Cell using process oriented techniques and comparing these with similar algorithms using traditional techniques for multiprocessing such as OpenMP. A number of classical computational problems have already been solved in occam. Should the sources to these be available, it would prove interesting to port these to occam-pi and adapt them for the Cell BE, such as a parallel Jacobi Iterator [CH91].

7.3 Long-term future work

It will be interesting to evaluate the continued use of the occam-pi language as new computational platforms emerge. According to reports, the microprocessor and graphics processor industries are converging to some degree. New architectures like the Cell BE, Nvidias CUDA, or in the future Intel's Larabee, are starting to have more features in common. Desktop processors are gaining more vector processing/streaming functions with wider buses and increased concurrency, while graphics processors are getting better support for branching and memory management which have been lacking to date.

Extending the occam-pi language to support such features is necessary in the long run if it is to remain a useful language. As such it is important to continue developing compiler support for new platforms.

7.3.1 Support for CUDA/GPU/Larabee-like architectures

The Transterpreter is designed to be portable across architectures that support the C programming language. Using the Transterpreter it may be possible to develop a general purpose, cross-platform parallel programming system, to support systems ranging from embedded systems to large NUMA based machines and clusters, as well as specialised hardware, such as the Cell or CUDA-like and Larabee-like systems. Ideally such a system would allow different architectures to interoperate, much like the Transterpreter currently allows the heterogeneous processors of the Cell BE interoperate, not requiring

separate user programs to be compiled.

7.3.2 Tock support for the Transterpreter as a back-end

With the potential addition of CIF support for the Transterpreter, thereby increasing compatibility with CCSP, the Transterpreter could become a back end for programs compiled by Tock. For example, having CIF support in the Transterpreter would allow Tock to target a wider range of platforms, such as the Cell BE, giving user programs both the added performance of C as well as the safety that the compiler provides. Alternatively it would be possible to add a transputer byte code back end to the Tock compiler. This would enable programs compiled by Tock to run on all of the platforms that the Transterpreter supports.

7.3.3 Compiler support for generating code for the Cell

Compiler support could be added to Tock or OCC21 to make it aware of the underlying architecture of the Cell BE. The compilers could perform correctness checks on all inter-processor channel communication. Currently the compilers are not aware of inter-processor channels, and these are not subject to any safety checks that could otherwise be inferred by the compiler.

Support for designating processes, or a process networks locality could be added to Tock, with additional support in the Transterpreter runtime. Similar syntax to the original Transputer `PLACED PROCESS` could be used. This could be a useful additional feature making it easier for a compiler to statically analyse inter-processor channel communication. This could also enable more efficient compilation of user programs, resulting in more efficient use of memory on each SPU as only the portions of the byte code that are required would need to be copied to them.

7.3.4 Compiler support for SIMD processing

In line with supporting the new features available on modern processors, SIMD code generation support could be added to Tock. For example, this could be based on loop-level inference using occam's PAR syntax as described in OEP 149 [Dim06]. This would be useful in many common algorithms on most modern CPU architectures and, at some point in the future, GPUs.

Bibliography

[ANL09] Argonne National Laboratory. The MPICH2 project page, 2009. Available from: `http://www.mcs.anl.gov/research/projects/mpich2/`.

[Bar00] Frederick R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press.

[Bar01] Frederick R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Majid Mirmehdi, Alan Chalmers, and Henk Muller, editors, *Communicating Process Architectures 2001*, number 59 in Concurrent Systems Engineering Series, pages 265–282. IOS Press, Amsterdam, The Netherlands, September 2001.

[Bar03a] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent at Canterbury, June 2003.

[Bar03b] Frederick R.M. Barnes. Window Gadgets and WG Builder, 2003. Available at: `http://www.frmb.org/wg.html`.

[Bar05] Frederick R.M. Barnes. Interfacing C and occam-pi. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 249–260, IOS Press, The Netherlands, September 2005. IOS Press.

[BCG+05] Geoff Biggs, Toby Collett, Brian Gerkey, Andrew Howard, Nate Koenig, Jordi Polo, Radu Rusu, and Richard Vaughan. The Player/Stage project, 2005. Available from: `http://playerstage.sourceforge.net/`.

[BDK+05] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Stephen S. Pawlowski, and Justin R. Rattner. Platform 2015: Intel ® processor and platform evolution for the next decade. Technical report, Intel Corporation, 2005.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[Bea96] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*, page 15, Berkeley, CA, USA, 1996. 4th annual Tcl/Tk workshop, USENIX Association.

[Bea97] David M. Beazley. Feeding a large-scale physics application to Python. In *6th International Python Conference*. San Jose, California, 1997.

[Bea05] David M. Beazley et al. SWIG-1.3 Documentation. Technical report, University of Chicago, 2005. Available from: `http://www.swig.org/Doc1.3/index.html`.

[Bel73] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[BJV03] Frederick R.M. Barnes, Christian Jacobsen, and Brian Vinter. RMoX: A Raw-Metal occam Experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent*

Systems Engineering, pages 269–288, IOS Press, Amsterdam, The Netherlands, 2003. IOS Press.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press.

[BOS+96] Andrè W. P. Bakkers, G. W. Otten, M. H. Schwirtz, R. Bruis, and Jan F. Broenink. Implementation of KRoC on Analog Devices SHARC DSP. In Brian C. O'Neill, editor, *Proceedings of WoTUG-19: Parallel Processing Developments*, pages 179–190, Feb 1996.

[Bra85] Stephen Brain. Implementing the occam portakit. *The Computing Magazine*, 22, July 1985.

[Bro06] Neil C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, September 2006.

[Bro07] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, July 2007.

[BW02] Frederick Barnes and Peter Welch. Prioritised dynamic communicating processes: Part 1. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, volume 60 of *Concurrent Systems Engineering*, pages 321–352, IOS Press, Amsterdam, The Netherlands, September 2002. IOS Press.

[BW03a] Frederick R.M. Barnes and Peter H. Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings-Software*, 150(2):121–136, April 2003.

[BW03b] Neil C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, Sept 2003.

[BW04] Frederick R.M. Barnes and P.H. Welch. Communicating Mobile Processes. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *WoTUG-27, Concurrent Systems Engineering*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press.

[BWS05] Frederick R.M. Barnes, Peter H. Welch, and Adam T. Sampson. Barrier synchronisation for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA Press.

[CH91] R. D. da Cunha and T. Hopkins. Parallel overrelaxation algorithms for systems of linear equations. In *Proceedings of the world transputer user group (WOTUG) conference on Transputing '91*, pages 159–169, Amsterdam, The Netherlands, The Netherlands, 1991. IOS Press.

[DBB+93] E.D. Dickmanns, R. Behringer, C. Brudigam, D. Dickmanns, F. Thomanek, and V. von Holt. An All-Transputer Visual Autobahn-Autopilot/Copilot. In *ICCV93*, pages 608–615, 1993.

[DDHS00] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[DHWN94] Mark Debbage, Mark Hill, Sean Wykes, and Denis Nicole. Southampton's portable occam compiler — SPOC. In *Proceedings of WoTUG-17: Progress in Transputer and occam Research, volume 38 of Transputer and occam Engineering*, pages 40–55. IOS Press, 1994.

[Dim05] Damian J. Dimmich. occam-pi module for SWIG resource page, 2005. Available at: `http://projects.cs.kent.ac.uk/projects/swigoccam/trac/`.

[Dim06] Damian J. Dimmich. Vector operations, 2006. Available from: `http://www.cs.kent.ac.uk/research/groups/sys/wiki/OEP/149`.

[DJ05] Damian J. Dimmich and Christan L. Jacobsen. A Foreign Function Interface Generator for occam-pi. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, pages 235–248, Amsterdam, The Netherlands, September 2005. IOS Press.

[DJJ06] Damian J. Dimmich, Christian L. Jacobsen, and Matthew C. Jadud. A cell transterpreter. In Peter Welch, Jon Kerridge, and Frederick Barnes, editors, *Communicating Process Architectures 2006*, Concurrent Systems Engineering Series. IOS Press, Amsterdam, September 2006.

[DJJS] Damian J. Dimmich, Christian L. Jacobsen, Matthew C. Jadud, and Jonathan Simpson. Robodeb virtual appliance. Available from: `http://www.vmware.com/appliances/directory/338`.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[Eic06] A. E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine achitecture. *IBM Systems Journal*, 45:59–84, January 2006.

[Ent06] Sony Computer Entertainment. Playstation III Website, 2006. Available from: `http://www.us.playstation.com/PS3`.

[Fla01] Matthew Flatt. PLT MzScheme: Language manual. Reference Manual PLT-TR2001-1-v103p1, PLT Scheme Inc., August 2001. `http://`

plt-scheme.org/techreports/. Available from: http://download.plt-scheme.org/doc/103p1/pdf/mzscheme.pdf.

[Fla05] B. Flachs et al. A Streaming Processing Unit for a CELL Processor. Technical report, ISSCC - Digest of Technical Papers, Session 7 / Multimedia Processing / 7.4, 2005.

[For00] Formal Systems (Europe) Ltd. *FDR2 User Manual.* 3, Alfred Street, Oxford, OX1 4EH, UK., May 2000.

[GCC06a] GCC Team. Gnu compiler collection homepage, 2006. Available from: http://gcc.gnu.org/.

[GCC06b] GCC Team. Using vector instructions through built-in functions, 2006. Available from: http://gcc.gnu.org/onlinedocs/gcc-3.4.0/gcc/Vector-Extensions.html.

[Gro04] LEGO Group. The LEGO Mindstorms homepage, 2004. Available from: http://www.legomindstorms.com/.

[Gro08] Khronos OpenCL Working Group. Khronos Group Inc., 2008.

[GTK05] GTK+ Team. GTK (The Gimp Toolkit) website, 2005. Available from: http://www.gtk.org/.

[Hig97] Julian Highfield. A Transputer emulator, 1997. Available from: http://spirit.lboro.ac.uk/emulator.html.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hyd95] Daniel C. Hyde. *Introduction to the Programming Language Occam*. Department of Computer Science, Bucknell University, Lewisburg, PA 17837, USA, March 1995.

[IBM05] IBM. Libspe v1.0 documentation, 2005. Available from: http://www.bsc.es/projects/deepcomputing/linuxoncell/stable/release3.0/%libspe/libspe_v1.0.pdf.

[IBM06] IBM. IBM Full-System Simulator for the Cell Broadband Engine Processor, 2006. Available from: `http://www.alphaworks.ibm.com/tech/cellsystemsim`.

[IBM09] IBM. XL C/C++ for Multicore Acceleration for Linux, 2009. Available from: `http://www-01.ibm.com/software/awdtools/xlcpp/multicore/`.

[INM84a] INMOS Limited. *The occam Portakit Implementors Guide*. Bristol, November 1984.

[INM84b] INMOS Limited. *occam Programming Manual*. Prentice Hall Trade, 1984.

[INM84c] INMOS Limited. *occam2 Reference Manual*. Prentice Hall, 1984.

[INM88] INMOS Limited. *Transputer instruction set: a compiler writer's guide*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1988.

[INM95] INMOS Limited. occam 2.1 Reference Manual. Technical report, INMOS Limited, May 1995. Available at: `http://www.wotug.org/occam/`.

[INM10] INMOS Limited. occam test suite, 2010. Available from: `http://projects.cs.kent.ac.uk/projects/kroc/trac/browser/kroc/trunk/tes%ts/cgtests`.

[Int97] Intel Corporation. *The Complete Guide to Mmx Technology*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[Jac08] Christan L. Jacobsen. *A Portable Runtime for Concurrency Research and Application*. PhD thesis, University of Kent, 2008.

[JDC06] Christian L. Jacobsen, Damian J. Dimmich, and Matthew C.Jadud. Native Code Generation using the Transterpreter. In Peter Welch, Jon Kerridge, and Frederick Barnes, editors, *Communicating Process Architectures 2006*, Concurrent Systems Engineering Series. IOS Press, Amsterdam, September 2006.

[JG88] Geraint Jones and Michael Goldsmith. *Programming in occam 2*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.

[JJ04] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Prof. Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106. IOS Press, Amsterdam, September 2004.

[JJ05] Christian L. Jacobsen and Matthew C. Jadud. Towards concrete concurrency: occam-pi on the LEGO Mindstorms. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 431–435, New York, NY, USA, 2005. ACM Press.

[JJ07] C. L. Jacobsen and M. C. Jadud. Concurrency, robotics and robodeb. Technical report, AAAI Press, Menlo Park, California, March 2007.

[JJD06] Matthew C. Jadud, Christian L. Jacobsen, and Damian J. Dimmich. Concurrency on and off the sensor network node. *SEUC 2006 workshop*, June 2006.

[JL92] Doug Brown John Levine, Tony Mason. *lex & yacc*. O'Reilly, 1992.

[JS07] Ulrik Schou Jorgensen and Espen Suenson. trancell - an Experimental ETC to Cell BE Translator. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 287–298, jul 2007.

[JSJ08] Matthew C. Jadud, Jon Simpson, and Christian L. Jacobsen. Patterns for programming in parallel, pedagogically. *SIGCSE Bull.*, 40(1):231–235, 2008.

[JW92] G. R. Justo and Peter H. Welch. Serialisation as a paradigm for the engineering of parallel programs. In *PARLE '92: Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 975–976, London, UK, 1992. Springer-Verlag.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research & Development*, 49(4/5):589–604, July/September 2005.

[Kil96] Mark Kilgard. GLUT - The OpenGL Utility Toolkit, 1996. Available from: `http://www.opengl.org/resources/libraries/glut/`.

[KPP06] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, 2006.

[Kri08] Mads Alhof Kristiansen. Cellcsp source code, March 2008. Available from: `http://code.google.com/p/cellcsp/`.

[Kri09] Mads Alhof Kristiansen. Communicating Sequential Processes on the Cell Broadband Engine, January 2009.

[Lan05] Sam Lantiga. libSDL - Simple Directmedia Layer, 2005. Available from: `http://www.libsdl.org/`.

[Lee06] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[Lew96] C. S. Lewis. OCINF - The Occam-C Interface Generation Tool. Technical report, Computing Laboratory, University of Kent, Canterbury, 1996.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM Press.

[Mil99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[Mob09] MobileRobots Inc. Pioneer III robotics platform, 2009. Available from: `http://www.activrobots.com/ROBOTS/p2dx.html`.

[Moo99] James Moores. CCSP - A Portable CSP-Based Run-Time System Supporting C and occam. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 147–169, mar 1999.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes – parts I and II. *Journal of Information and Computation*, 100:1–77, 1992. Available as technical report: ECS-LFCS-89-85/86, University of Edinburgh, UK.

[Mue05] S. M. Mueller et al. The Vector Floating-Point Unit in a Synergistic Processor Element of a CELL Processor. *IEEE Symposium on Computer Arithmetic, 2005. ARITH-17.*, 17:59–67, 2005.

[MWS99] Tom Davis Mason Woo, Jackie Nieder and Dave Schriener. *OpenGL Programming Guide, Third Edition*. Addison Wesley, Reading, Massachusetts, third edition, 1999.

[NF08] Andreas Gal Albert Noll and Michael Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. In *Workshop on Cell Systems and Applications*, Beijing, China, June 2008.

[Nok09] Nokia Inc. S60 official site, 2009. Available from: `http://www.s60.com/life`.

[occ09] occam-pi.org. occam-pi library documentation generated using OccamDoc, 2009. Available from: `http://occam-pi.org/occamdoc/`.

[ocml07] occam-com mailing list. occam-com mailing list, 2007. Available from: `http://www.occam-pi.org/list-archives/occam-com/`.

[OIS+06] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI Microtask for programming the Cell Broadband Engine processor. *IBM Syst. J.*, 45(1):85–102, 2006.

[Ope08] Open Watcom. Open watcom compiler, 2008. Available from: `http://www.openwatcom.org`.

[PBBL07] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta. Cellss: making it easier to program the cell broadband engine processor. *IBM J. Res. Dev.*, 51(5):593–604, 2007.

[Pha05] D. Pham et al. The Design and Implementation of a First-Generation CELL Processor. *Digest of Technical Papers*, pages 184–185, February 2005.

[Poo96] Michael D. Poole. Occam for all - Two Approaches to Retargetting the INMOS Compiler. In Brian C. O'Neill, editor, *Proceedings of WoTUG-19: Parallel Processing Developments*, pages 167–178, feb 1996.

[Poo98] M.D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering*, Address, April 1998. WoTUG, IOS Press.

[Pug92] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:4–13, 1992.

[Rat05] Justin Rattner. Multi-core to the masses. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[RS08] Carl G. Ritson and Jonathan Simpson. Virtual Machine Based Debugging for occam-pi. In Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R.M. Barnes, Alistair A. McEwan, Gardner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 293–307, Amsterdam, The Netherlands, September 2008. IOS Press.

[RSB09] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION 2009: Coordination Models and Languages, 11th International Conference, Proceedings*, Lecture Notes in Computer Science, pages 163–183. Springer, June 2009.

[RSJ04] Kalla R, Balaram Sinharoy, and Tendler J.M. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40– 47, March 2004.

[RWW91] Herman Roebbers, Peter Welch, and Klaas Wijbrans. A generalized FFT algorithm on transputers. Technical report, in D. L. Fielding (ed), Transputer Research and Applications 4, IOS, 1991.

[Sam06] Adam T. Sampson. Occam Enhancment Proposals (OEP), 2006. Available from: `http://www.cs.kent.ac.uk/research/groups/sys/wiki/OEP`.

[Sam09a] A.T. Sampson. Occade - arcade game library for education, 2009. Available from: `https://www.cs.kent.ac.uk/research/groups/sys/wiki/Occade`.

[Sam09b] A.T. Sampson. Occade - arcade game library for education documentation, 2009. Available from: `http://occam-pi.org/occamdoc/occade.html`.

[SARW98] Tim Sheen, Alastair R. Allen, Andreas Ripke, and Stacy Woo. oc-X: an Optimising Multiprocessor occam System for the PowerPC. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 167–186, mar 1998.

[SB08a] A.T. Sampson and N.C.C. Brown. Generics in Small Doses: Nanopass Compilation with Haskell. Technical report, 2008.

[SB08b] A.T. Sampson and N.C.C. Brown. Tock - translator from occam to C from Kent, 2008. Available from: `http://projects.cs.kent.ac.uk/projects/tock/trac/`.

[SBW03] Mario Schweigler, Frederick Barnes, and Peter Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In Jan F Broenink and Gerald H Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 199–224, Amsterdam, The Netherlands, September 2003. IOS Press.

[SCS+09] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Sugerman, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *IEEE Micro*, 29(1):10–21, 2009.

[Sha06] Stephen Shankland. IBM aims to tame its Cell chip, 2006. Available from: `http://www.zdnetasia.com/news/hardware/0,39042972,39314661,00.htm`.

[SJ08] Jonathan Simpson and Christian L. Jacobsen. Visual Process-oriented Programming for Robotics. In Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R. M. Frederick Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 365–380, Amsterdam, The Netherlands, September 2008. IOS Press.

[SJJ06] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. Mobile Robot Control: The Subsumption Architecture and occam-pi. In Peter H. Welch, Jon M. Kerridge, and Frederick R.M. Barnes, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pages 225–236, Amsterdam, The Netherlands, September 2006. IOS Press.

[SJJ07] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. A Native Transterpreter for the LEGO Mindstorms RCX. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 339–348, Amsterdam, The Netherlands, September 2007. IOS Press.

[SO98] Marc Snir and Steve Otto. *MPI - The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[Ste05] Greg Stein. Python at Google, March 2005. Available from: `http://www.sauria.com/~twl/conferences/pycon2005/20050325/Python%20at%2%0Google.notes`.

[Sto03] Jonathan Stott. WGoccam: GUI and Graphics Libraries for occam. Technical report, Computing Laboratory, University of Kent, Canterbury, 2003.

[SVP07] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Programming the Cell Processor, 2007. Available from: `http://www.ddj.com/hpc-high-performance-computing/197801624` [cited 2008].

[SWB05] A.T. Sampson, P.H. Welch, and Frederick R.M. Barnes. Lazy cellular automata with communicating processes. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, Amsterdam, The Netherlands, September 2005. IOS Press.

[SWD05] Dipanwita Sarkar, Oscar Waddell, and Dybvig. Educational pearl: A nanopass framework for compiler education. *Journal of Functional Programming*, 15(05):653–667, 2005. Available from: `http://journals.cambridge.org/action/displayAbstract?fromPage=online\&a%id=331529`.

[Sys09] Program Verification Systems. VivaMP Tool, 2009. Available from: `http://www.viva64.com/vivamp-tool/`.

[tun05] Theory Underpinning Nanotech Assemblers, January 2005. EPSRC grant EP/C516966/1. Available from: `http://www.cs.york.ac.uk/nature/tuna/`.

[Van08] Sriram Vangal. An 80-Tile Sub-100-w TeraFLOPS Processor in 65-nm CMOS. volume 43, pages 29–41, 2008.

[Vel98] Kevin Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, University of Kent at Canterbury, December 1998.

[Vin07] Brian Vinter. Extreme Multiprogramming Course, 2007. Available from: `http://www.diku.dk/hjemmesider/ansatte/vinter/xmp/`.

[VMw] VMware. VMware Player. Available from: `http://www.vmware.com/products/player/`.

[WB01] Peter H. Welch and Frederick R.M. Barnes. Mobile Data Types for Communicating Processes. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications 2001*, volume 1, pages 20–26. CSREA, CSREA Press, June 2001.

[WB05] Peter H. Welch and Federick R.M. Barnes. Mobile Barriers for occam-pi: Semantics, Implementation and Application. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 289–316, IOS Press, The Netherlands, September 2005. IOS Press.

[Wel09] P.H. Welch. CO631 Concurrency Design and Practice course, 2009. Available from: `http://www.cs.kent.ac.uk/CO631`.

[Whi00] Paul White. Drawing Manager for Graphical occam. Project Report. Technical report, Computing Laboratory, University of Kent, Canterbury, 2000.

[Wik07] Wikipedia. Wikipedia entry on the Transputer, 2007. Available from: `http://en.wikipedia.org/wiki/Transputer`.

[WM99] D.C. Wood and J. Moores. User-Defined Data Types and Operators in occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 121–146, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[WMBW00] Peter H. Welch, J. Moores, Frederick R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[WNGG] Kevin Williams, Albert Noll, Andreas Gal, and David Gregg. Optimization strategies for a Java virtual machine interpreter on the Cell Broadband

Engine. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 189–198, New York, NY, USA. ACM.

[Woo98] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent, Canterbury, August 1998.

[WP97] Peter H. Welch and Michael D. Poole. occam for Multi-Processor DEC Alphas. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 152–174, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

[WSO+06] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the Cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[WSST08] P.H. Welch, S. Stepney, A.T. Sampson, and J. Timmis. CoSMoS Complex Systems Modelling and Simulation Infrastructure, 2008. Available from: `http://www.cosmos-research.org`.

[WW96] D. Wood and P. Welch. The Kent Retargetable occam Compiler. In B. O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47, pages 143–166. IOS Press, Netherlands, 1996.