SCALABLE SUPPORT
FOR
PROCESS-ORIENTED PROGRAMMING

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Carl G. Ritson

2013

Abstract

Process-oriented programming is a method for applying a high degree of concurrency within software design while avoiding associated pitfalls such as deadlocks and race hazards. A process-oriented computer program contains multiple distinct software processes which execute concurrently. All interaction between processes, including information exchange, occurs via explicit communication and synchronisation mechanisms. The explicit nature of interaction in process-oriented programming underpins its ability to provide manageable concurrency. These interaction mechanisms represent both a potential overhead in the execution of process-oriented software and a point of mechanical sympathy with emerging multi-core computer architectures. This thesis details engineering to reduce the overheads associated with a process-oriented style of software design and evaluate its mechanical sympathy.

The first half of this thesis provides an in-depth review of facilities for concurrent programming and their support in programming languages. Common concurrent programming facilities are defined and their relationship to process-oriented design established. It contains an analysis of the significance of mechanical sympathy in programming languages, trends in hardware and software design, and relates these to process-oriented programming.

The latter part of this thesis describes techniques for the compilation and execution of process-oriented software on multi-core hardware so as to achieve the maximum utilisation of parallel computing resources with the minimum overhead from process-oriented interaction mechanisms. A new runtime kernel design for the occam-pi programming language is presented and evaluated. This design enables efficient cache-affine work-stealing scheduling of processes on multi-core hardware using wait-free and non-blocking algorithms. This is complemented by modern compilation techniques for occam-pi program code using machine independent assembly to improve performance and portability, and methods for debugging the execution of process-oriented software using a virtual machine interpreter. Through application, these methods prove the mechanical sympathy and parallel execution potential of a process-oriented software.

Acknowledgements

I began this PhD while working as a research associate on the RMoX project under the direction of Fred Barnes. My work owes much to Fred Barnes as both my supervisor and the author of much of the prior work and computer code on which this thesis builds.

My research has been supported financially by the EPSRC funded projects RMoX (EP/D061822/1) and CoSMoS (EP/E049419/1). I have also received financial support from the University of Kent's staff development fund and the School of Computing.

The community of researchers interested in concurrency at Kent was a significant motivation to me and I am sad to see that that the Concurrency Research Group has all but ceased to exist. Peter Welch and his enthusiasm has been the hub for much concurrency research at Kent. He inspired and encouraged me to pursue research in this area. Christian Jacobsen, Matt Jadud and Damian Dimmich's work on the Transterpreter provided an excellent test bed for many of my ideas and good source of code for me to rewrite. Adam Sampson's input and feedback on my work has proved invaluable. Jon Simpson with whom I have shared an office and the path of PhD student has always been a good source of moral support and pragmatism.

Utmost I would like my family, close friends and especially Jasmine. All of whom have always been there offering their support.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

This thesis outlines and demonstrates techniques for the compilation and execution of process-oriented software on multi-core hardware, with an emphasis on efficient use of computer resources (processor time and memory space). This work addresses two connected research questions:

1. Can software which uses unbounded concurrency for structure be efficient?

2. Can process-oriented programming be used to build scalable software?

The first question applies when software uses a high degree of concurrency without regard for the availability of parallel processing resources. The second question investigates the effect of making more parallel processing resources available to such software.

We use *concurrency* to describe where multiple computations *may* happen at the same time. This is a disambiguation from *parallelism* where multiple computations *do* happen at the same time. Process-oriented programming is a method for applying a high degree of concurrency within software design [228]. It typically produces many more concurrent program elements than there are parallel computing resources to execute simultaneously.

A process-oriented computer program contains multiple isolated software processes which execute concurrently. As processes are isolated from each other all interaction, including information exchange, occurs via explicit communication and synchronisation mechanisms. The explicit nature of interaction in process-oriented programming underpins its ability to manage complexities arising due to concurrency, such as non-determinism. However these interaction mechanisms represent a potential overhead in the execution of process-oriented software.

Computer hardware is moving from systems with a single programmable computation element or *core*, to multiple programmable parallel computation elements or *multi-core*. A high degree of concurrency is critical to utilising the parallel computing resources made available by multi-core computer processors. If my work consists of many tasks I *can* do at the same time (independent tasks) then given a clone of myself I will be able to *do* all my tasks in half the time.

Commodity computer hardware, programming tools and operating systems do not explicitly support process-oriented programming. This means the basic operations on which process-oriented programs depend have significant overheads. The implication is that process-oriented software is not as efficient as other programming methods and may not be able to achieve equal or superior performance.

In the context of this thesis, *mechanical sympathy* describes the congruence of programming models and computer hardware. A mechanically sympathetic computer program uses computer hardware in efficient manner; progress towards a solution is taken along the path of least resistance. A mechanically sympathetic programming language embeds, or encourages the programmer to use, a programming model which has an efficient mapping to real computer systems.

In this thesis I seek to demonstrate the potential of process-oriented software, by minimizing the amount of processor time, memory space and bandwidth consumed in overheads while achieving the maximum use of available parallel processing facilities. This is important as process-oriented programs have a high degree of mechanical sympathy with present and emerging multi-core computer systems (2.7.5). Additionally these benefits can be exploited by other concurrent programming methods through functional decompositions presented in this thesis (2.4.1) and in related work [228].

1.1 Problem Statement

In order to fully utilise the parallel processing capabilities of modern computer hardware, computer software must utilise a higher degree of concurrency. *Scalable software* is software which has increased performance, faster execution or more work completed per unit time, when additional parallel computing resources are made available to it. The synchronisation and scheduling overheads of common concurrency mechanisms mean that in order to build scalable software the typical concurrent work unit (grain-size [176, 209]) must be relatively large. For example amortizing the cost of a 10ms synchronisation, reducing it below 5% of execution time, requires a work unit lasting at least 200ms.

By reducing the overheads associated with a class of concurrency mechanisms, programmers will be able to introduce more concurrent elements into software without adversely affecting performance. In turn, such software will make more efficient use of available hardware and have improved performance as a result. Better performance may be reduced execution time, reduced response time or comparable execution time with reduced power consumption as larger numbers of lower powered processing elements are substituted for fewer higher powered ones.

1.2 Limitations

This thesis does not address the issue of designing parallel software or computer algorithms. In this thesis I explain what is meant by process-oriented software and how this work allows process-oriented software designs to make use of multi-core hardware, but do not address the process-oriented design process. I would refer the reader to Adam

Sampson's thesis [228] which does much to explain process-oriented software design. Beyond process-oriented design, there is a large body of work on developing parallel algorithms, such as Ian Foster's work [110].

1.3 Contributions

This thesis makes the contributes the following research contributions:

D1. **Message-passing between concurrent components can be at least one order of magnitude faster than presently implemented in common programming languages or concurrency frameworks.** This includes modern multi-processor or multi-core computer hardware.

D2. **The parallel execution potential of software using concurrency for structure can be efficiently extracted.** This demonstrates that process-oriented programming can be used to build scalable software.

D3. **Process-oriented programming has a high degree of mechanical sympathy with modern multi-core computer hardware.** Mechanical sympathy is a key factor in the applicability, efficiency and adoption of programming languages.

This thesis also makes the following technical contributions:

T1. **Development of a highly efficient scheduler for process-oriented systems on multi-core hardware.** The scheduler is designed to support thousands of active processes on commodity hardware using runtime decision making based on heuristics rather than a priori knowledge of the system being scheduled.

T2. **Algorithms for synchronous channel communication with choice on multi-core hardware.** These enable communication between pairs of processes using wait-free scheduler interaction and permitting choice in one of the communicating parties.

T3. **Algorithms for choice across channels and timers on multi-core hardware.** These allow a process to make a choice over a number of communication channels and a system timer.

T4. **Algorithms for supporting explicit priority groups.** These scheduler extensions support priority groups in order to guarantee execution of high priority processes before low priority processes.

T5. **Algorithms for barriers on multi-core hardware with optimisations for cache-coherence and priority.** These provide synchronisation between tens of thousands of software processes while reducing the negative cache impact of access to a heavily contended resource.

T6. **Additional performance for the occam-pi programming language through the application of machine independent assembly (LLVM).** This provides direction for the compilation of future process-oriented languages.

T7. **Development of virtual machine debugging techniques for process-oriented programming.** These provide runtime support for introspective analysis of process-oriented programs.

1.4 Road Map

Chapter 2 examines the motivations for the use of concurrency in software design and support for concurrency in programming languages. This can be seen as a review of related work in concurrent programming techniques and a definition of process-oriented programming. It contains a history and analysis of hardware support for concurrency and the development of programming techniques for managing concurrency. It also establishes tenants of contribution *D3* (1.3).

Chapter 3 documents the development of a multi-core scheduler for process-oriented software. It contains algorithms for implementing core elements of process-oriented software such as communication and synchronisation on multi-core systems. These relate to contributions *D1* through *D3* and *T1* through *T5* (1.3).

Chapter 4 describes methods for compiling and executing process-oriented software on modern computer systems. The focus is on the occam-pi programming language and translating it to machine-independent assembly languages. This relates to contribution *T6* (1.3).

Chapter 5 explores potential debugging and tracing techniques for procress-oriented programming. These make use of a virtual machine implementation of occam-pi, such as that provided by the Transterpreter [146]. This relates to contribution *T7* (1.3).

Chapter 6 summarises the conclusions of this thesis and discusses some directions for future research. Attention is given to the direction of current computer software and hardware development.

Chapter 2

Concurrency

This chapter is an in-depth survey of concurrency support in programming languages, its origins and trends. The content of this chapter is intended to be accessible to a wide audience, with many sections essentially readable without much, if any, prerequisite knowledge. To support this, programming concepts are, where appropriate, analogised to examples from other domains, for example cookery.

A key purpose of this chapter is to document and analyse a trend towards data-oriented methods in the mechanisms provided for programming concurrency by popular programming languages (see 2.2 and 2.7.2). This trend is rooted in the shared ancestory of popular programming languages, an ancestory which eschews a model for programming concurrency (see 2.7.1). Recent trends in hardware parallelism have motivated the need for mechanisms for programming concurrency (see 2.1.3 and 2.6). This thesis argues that process-oriented programming satisfies the need for a mechanically sympathetic model of programming concurrency on new and emerging computer hardware (see 2.7.5). Furthermore, it argues that process-oriented programming is more appropriate than data-oriented methods while still maintaining support for existing data-oriented programming primitives (see 2.4.1).

This chapter is broad in scope and as such the first-time reader may wish to omit some sections. For the purpose of understanding other chapters the material on paradigms in section 2.2 and in particular the message-passing paradigm (2.2.2) is essential. From the material in section 2.3, section 2.3.1 on processes, section 2.3.2.4 on messages and section 2.3.3 on choice should be read. The full definition of process-oriented programming in section 2.4 is also essential to this thesis. Finally, the memory synchonisation cost data given in section 2.6.1 may prove useful to understanding the arguments in chapter 3.

Road Map

Section 2.1 explores two key motivations for the use of concurrency in the design of computer programs: *expression* and *performance*. Expression describes the use of concurrency to address the parallel nature of the world. Performance refers to the use of concurrency to utilise parallel computer hardware. This section also considers recent and on-going trends in computer hardware to increasingly parallel computer systems. These key elements motivate the importance of research on concurrent programming languages and their run-time support.

Section 2.2 describes the main paradigms (styles) of concurrent programming: data-oriented concurrency and message-passing concurrency. Adding to this, section 2.3 details common language operations and primitives for handling concurrency. The intention is to provide a broad overview of distinct methods for introducing and handling concurrency in computer programs. These provide reference for later sections and chapters.

Section 2.4 defines process-oriented programming, a specific form of message-passing concurrency. Critically, the primitives surveyed in section 2.3 are given process-oriented definitions. This section establishes broadly that process-oriented programming is capable of concisely expressing other forms of concurrency. This is necessary to establish the wider applicability of work in this thesis.

Section 2.5 surveys support for concurrent programming in a broad range of popular and general-purpose programming languages. This section provides an integral reference for other sections; however, as a reference, cover to cover reading is not required. The subsequent section, section 2.6, gives an overview of support for concurrent programming in computer hardware. The subsection *Synchronisation* (2.6.1) identifies the costs of using shared memory for inter-processor synchronisation. The subsection *Interconnects* (2.6.2) describes the reasons behind the growth in synchronisation costs in modern computer hardware. These motivate the need to avoid inter-processor communication by using paradigms such as process-oriented programming and appropriate algorithm design as in chapter 3.

Section 2.7 analyses information presented in sections 2.5 and 2.6. The general trend is for minimal support for concurrent programming with a focus on a data-oriented style where support is present. This contrasts with process-oriented programming which uses concurrency as a method for structuring computer software. This trend is analysed with respect to popular programming languages and a critical aspect of programming language design, *mechanical sympathy*, is identified. Finally, key properties of a process-oriented style of programming are highlighted in relation to new and emerging computer architectures. It is postulated that process-oriented software is mechanically sympathetic with trends in hardware development.

2.1 Motivations

The motivations for the use of concurrency in the design and implementation of computer software can be loosely categorised into *expression* and *performance*.

Inherent in the creation of a computer program is the act of modelling a problem domain and mapping between it and a physical computer system. The programmer (or designer) produces a model, an abstraction, of a problem for the computer to solve; this is then expressed such that a computer can produce results or enact a solution. This idealised process can be seen in figure 1. The results produced by the computer are related to the problem domain in a manner determined by the programmer and the quality of the model produced by the designer. In reality the designer and programmer are

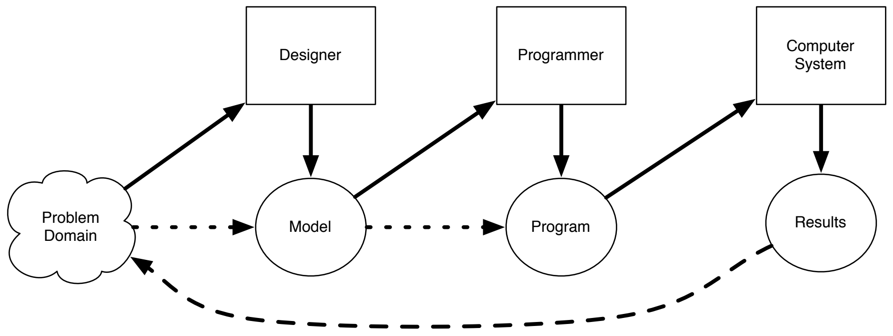


Figure 1: Idealised view of abstraction between problem domain and computer system.

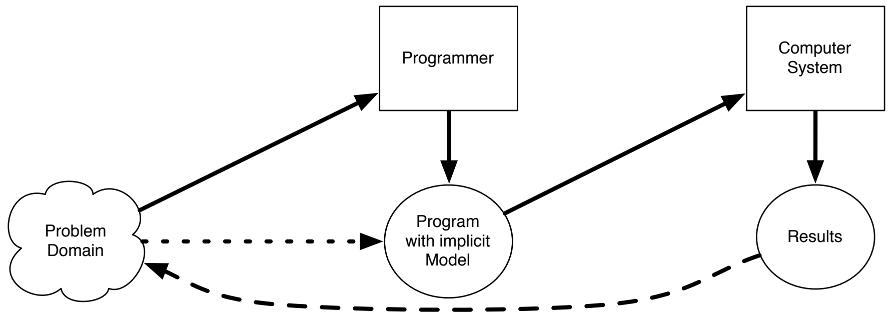


Figure 2: Simplified view of abstraction reflecting the reality that programs implicitly derive and embed domain models.

typically the same person, and the modelling step is implicit in the act of programming as in figure 2. That is to say the programmer does not consciously consider the model chosen or its implications. This can be problematic, in particular when the results of the program are to be scientifically rigorous [34].

Very early electronic computers such as ENIAC were programmed by the manipulation of switches and the physical connections between components. This soon gave way to the input of computer programs in the form of instructions typically stored on punched cards. The programmer expressed the problem directly in rudimentary machine operations, *assembly language*, e.g. load *a*, load 4, add, store as *b*. Essentially there was no abstraction between program and computer operations.

It takes considerable effort to express complex problems in terms of rudimentary *low-level* computer operations. This lead to the development of *high-level* programming languages. The first of these to see widespread usage was IBM's *FORTRAN* [42]. A key concern in the development of early high-level programming languages was to allow algebraic expressions. A program within the computer translates high-level expressions such as $b = a + 4$, into low-level assembly code. This program is called a *compiler*.

The introduction of early high-level languages, particularly FORTRAN, sees the introduction of commercial (*industry*) interests into computer programming. High level languages make computer programming more accessible, for example algebraic expression is more widely understood than the decomposition of problems into assembly language. This allows more people to write computer programs and increases the efficiency of existing programmers However, the compiler must be efficient. The assembly produced by the compiler must have similar performance, in terms of memory space and computational time requirements, as that produced by a skilled programmer. High level languages before FORTRAN were not efficient enough to justify their commercial use [43]. Early high-level languages might slow a computer down by a factor of five, an unacceptable slow down when some of the most basic computers cost upward of £200,000 a year to rent [1].

High level programming languages allow program text, *source code*, to more closely reflect and express the problem model. A key motivating factor in the development of most programming languages is the ability to clearly express a given problem domain, so that the program can accurately capture the model. This is particularly true of high-level languages developed in the late 1950s and early 1960s. Notable contemporaries to FORTRAN stand out as examples of this:

- *ALGOL* was developed by academia to address what were seen as failings in FORTRAN. Essentially ALGOL focuses on expression of algorithmic computing

[1]Based on the rent of a basic IBM 1401 $30,000 in 1959 and taking into account inflation based on the consumer price index.

for research scientists [41].

- *COBOL* was developed as a government and industry effort to create a standardised language for business-oriented tasks [208]. It was heavily used in government and commercial projects, to such an extent that more than 40 years after its inception it is still the 24th most popular programming language by usage [246].
- *Lisp* was developed by academia to express computation in a manner consistent with the *lambda-calculus*, which permits reasoning about the computability of mathematical functions [177]. The *Church-Turing thesis* relates the lambda-calculus to *Universal Turing Machines* showing that both express the same functions, and that functions can express all arithmetically computable operations. This theory underpins the discipline of *functional programming* which Lisp embodies.

Assembly languages are generally described as second generation languages, with first generation referring to machine operations represented directly by signals within the computer, so called *machine code*. Languages such as FORTRAN and ALGOL are third generation languages. Much research in the 1960s focused on structuring programs in third generation languages. One of the most well-known and widely used developments of this era is a programming paradigm called *object-oriented programming* [66]. Starting with SIMULA in 1968 [86] and exemplified by Smalltalk in 1973 2.5.27, object-orientation structures a program as a set of interacting objects. Loosely, each object represents a concrete or abstract concept and has a defined interface of operations it can perform on the behalf of other objects. For example, a toaster object might provide an interface with *set temperature* and *toast* operations. The inner workings of the toaster are hidden from other objects in the system. This is known as *encapsulation*. In principle this allows objects to be built and tested in isolation before being put together to produce larger programs.

High-level languages provide functionality not directly available in machine hardware. These functions may be provided by a *runtime system* (*RTS*), a library of prepared code which integrates with a program as it executes, at *runtime*. The invocation of the runtime system is (at least in part) embedded as part of the translation from high-level syntax to machine instructions. This occurs without the programmer needing to be aware of it taking place. For example if a given computer processor does not support integer division the programming environment may embed a call to a library function whenever the programmer divides two integers. Whether the hardware supports the operation or not, the result of the higher-level programming statement is always the same (albeit the library function may take longer to execute).

The runtime system may provide a *virtual machine* which simulates an idealised computer and hides the details of any particular computer hardware. This simplifies the task of converting a programming language into machine instructions as it need only be converted to one type of machine, the virtual one. However making the virtual machine representation of the program execute with the same performance as a direct conversion from language to machine instructions is non-trivial [115, 61, 39].

From the 1970s through to the early 1980s research and development focused heavily on third generation languages. The C programming language was released in 1973 (2.5.4) and gains popularity for *systems programming*, the development of software which controls a computer's hardware and provides functionality to other programs. By the early 1980s C has two major descendents, C++ (2.5.6) and Objective-C (2.5.20) both of which add object-oriented extensions to the language. The 1970s also sees the development of fourth and fifth generation languages; both of these aim to further remove the programmer from the low-level detail of the computer. Fourth generation languages are predominantly data processing languages focused on business needs; a third generation language is used in a defined architecture such that programs can be combined and reused and support tools exists to manage data and programs together [171]. Fifth generation languages and logic programming languages, such as Prolog, remove the programmer from specifying a solution to a problem, instead the properties of a valid solution are specified and the computer provides valid solutions within these *constraints* [75].

Despite the significant effort invested in fourth and fifth generation programming language research and development they have gained very limited popularity [246]. This is an important observation that is addressed in the analysis presented in section 2.7.

The need to handle and exploit concurrency emerges almost as soon as the first electronic computers are put in to use and gains weight as a research topic as computers grow in scale and speed [83, 90, 109, 119, 106]. Computers are constructed from multiple concurrent components which do not all work at the same speed. Rapid advances in early electronic computers quickly exposed these operational asymmetries. This creates a need to *express* the concurrency present in the computer so that it can be managed. Further to this, with the relatively high cost of computers and associated overheads[2] prior to the introduction of microcomputers in the 1980s, it was desirable that all aspects of the computer be utilised as fully as possible. This requires that multiple concurrent activities take places to maximise *performance*.

2.1.1 Expression

Expressive power in programming languages is conceptually the same as in natural language. For example, consider the instruction “bake a sponge cake”, this is a high-level statement than relies on the knowledge of what it is to bake and what a sponge cake is and how the two can be related. This instruction can be decomposed into simpler components, e.g. cream together 100 grams of sugar and butter, beat in an egg, combine 100 grams of flour, place in a baking tin, place the tin in an oven at 180 degrees celsius for 25 minutes. Each of these simpler instructions relies on further conceptual and specialised knowledge. Decomposition of these instructions can be continued until a base level is reached, e.g. turn 90 degrees right, lift right arm 50cm, move right arm 30cm forward, grasp cupboard handle in right hand, move right arm backward 30cm, and so on. This is essentially *assembly language*. Each level of decomposition moves

[2]Wages for the team of programmers and administrators along with the cost of electricity.

from an abstract generalised solution to a concrete implemented solution. However successive levels of decomposition obscure the high-level intent; the entire sequence of operations must be read and considered in order to realise that the intent.

With respect to figure 1, if the problem domain contains concurrency then the designer can either include this in the model, or abstract it away. Support for expressing concurrency in the programming language, programming paradigm and its support tools, broadly the *programming system*, will affect the designer's decision in this respect. This is particularly true for implicit models as per figure 2.

Concurrency is generally present in any model which interacts with external entities. An external entity is anything outside the control of the computer. Critically this is a concern for any computer program that interacts with a user, as the user can potentially produce new input or behaviour at any point. This concern is often mitigated by only allowing user interaction at defined points, but this abstraction can reduce the quality of the interaction experienced by the user. In many cases this abstraction is not acceptable, computer games being one prime example. While playing a computer game, the player expects it to respond immediately to their input.

Above I defined an external entity as anything outside the control of the computer – this is an idealised model. In reality a computer consists of multiple distinct components all operating concurrently. Thus events concurrent to computation can originate from both inside and outside the computer. A specialised piece of software, the *operating system*, handles these internal and external events, passing them off to running programs as appropriate. While normal computer programs can ignore concurrency, it must be explicitly handled when developing the computer's operating system [244, 46].

Concurrency is also present in models which attempt to capture the behaviour and interactions of *real world* entities. For example, in a model of bird flocking, each bird observes the environment and decides its actions independently of and concurrently to other birds. This model can be programmed such that the behaviour of each bird is computed sequentially; however, expressing the model and programming it in this way is a specialisation which loses accuracy with respect to the model [35]. That is to say, readers of the program code, including the compiler, are unaware of the concurrency in the model. There is a danger that the implicit ordering introduced is relied on in such a way that affects results [136].

2.1.2 Performance

While expression relates to how a computer program captures the model, performance relates to how the program makes use of the computer hardware. Just as the problem domain can contain concurrency, if the computer has multiple programmable computation elements then the hardware also contains concurrency. If concurrency exists in the computer's hardware then it is desirable to use this to improve performance. Improved performance might mean reduced execution time as work is divided between elements, or reduced response time to events if background work is done on one element while others are kept available to respond to new events. Improved performance could also mean more efficient use of energy by dividing work between low-power computational elements as oppose to using a single high-power element (2.1.3) [196].

In order to take advantage of concurrency in the computer, concurrency must be introduced into the program. If the model does not include concurrency then this concurrency exists purely for performance. This is the case in the parallelisation of algorithms. A textbook example of this is the Mandelbrot set [228]. A model for generating a fractal image from the Mandelbrot set is to divide the image into pixels (points) identified by coordinates horizontal x and vertical y, and a computation is then performed for each pair of x and y to derive the colour of that pixel. As the order of these computations does not matter to the model, in principle they can be programmed to execute concurrently. This would allow the programming system to execute as many as possible in parallel making use of all available concurrent computation elements. The Mandelbrot set is an example of an *embarrassingly parallel* problem: a problem that can in principle use all available computation elements without any significant alterations to its model.

If concurrency exists in the model then it can be expressed in the program and thus the model utilises the computation elements directly. With reference to the bird flocking example in section 2.1.1, if the birds are concurrent components their behaviour can be computed in parallel by available parallel computation elements. Direct exposure of model concurrency to hardware concurrency may not always be appropriate for performance. The model may either have too little concurrency and must be parallelised or it may have too much concurrency and must be serialised. In this thesis I seek to address the latter of these concerns.

The programmer should not need to optimise out model concurrency to achieve performance. A program should capture all concurrency present in the model it expresses. Hence where additional hardware is available the program and hence the model can make full use of it. By extension if further performance is required the programmer should be free to parallelise without concern that their optimisations reduce the applicability of the program to systems where hardware is unavailable.

2.1.3 Timeliness

The work presented in this thesis is timely because computer hardware is moving from systems with a single programmable computation element or *core*, to multiple programmable parallel computation elements or *multi-core*. Historically a computer *processor*, central processing unit or CPU, only had a single core. I will use the word processor when the number of cores is not relevant.

The shift in design toward multi-core has been necessitated by the reaching of the physical limitations associated with making a processors go faster. In order to increase the processing speed of the processor either the clock frequency (rate of instruction execution) must be increased, or additional hardware must be added to increase the internal parallelism (executing multiple instructions simultaneously).

2.1.3.1 Energy Usage

The power consumption of an integrated circuit (and hence processor) is approximately proportional to the clock frequency [140]. This might lead one to believe that a processor can run twice as fast simply by doubling its clock frequency and consume only twice as much power in doing so. However, as clock frequency is increased the quality of the signals within the processor is degraded as there is less time for internal voltages to change and be appropriately recognised. To counter this degradation the signalling voltage within the processor can be increased, making the signals clear again. Ohm's Law tells us that power consumption is proportional to the square of the voltage. Furthermore as the processor is operating faster it requires more buffers to mediate between its internal speed and the speed of the rest of the hardware, which increase the size of the integrated circuit and hence its power consumption (via capacitance in Ohm's Law). Therefore if we increase the clock frequency by a factor of two, increase the voltage by 25% and add 25% more components to mediate the increased speed, the processor now consumes approximately four times as much power.

Electrical energy (power) consumed by a computer processor is ultimately converted to thermal energy (heat). The energy input moves electrons within the processor which encounter resistance (much like the friction we encounter pushing a stone along the ground). This resistance causes energy to be lost from the motion of the electrons and be converted into heat. If we increase the power input to the processor more heat is generated. This heat increases the internal temperature of the processor. At higher temperatures the particles which make up the processor have more energy and move more erratically. Computation within the processor relies on the predictable nature of the flows of electrons within. Beyond a certain temperature the processor ceases to function predictably due to the erratic motion of the electrons. Cooling can reduce the temperature and allow the processor to function, but cooling in turn requires more power and space. Fundamentally the maximum power we can effectively put in to a computer processor is bounded. In turn this limits the maximum clock frequency and the linear execution speed.

2.1.3.2 Parallelism

There are three approaches to adding parallelism to a processor: flow optimisation (out-of-order execution and associated techniques), single instruction multiple data (vector processing) and multiple instruction multiple data (multi-core). Michael Flynn's work provides a taxonomy of these architectures and rigorous analysis of the performance implications of each [109].

Internally a processor is divided into elements which perform different tasks, such as fetching data from memory or adding numbers together. Simplistically, flow optimisation such as out-of-order execution attempts to keep all of these elements busy doing meaningful work by looking ahead in the linear sequence of instructions and doing things ahead of time if the capacity is available. This has its limitations, foremost of which is that an instruction may be dependent on the result of an earlier instruction. The sequence of instructions can be thought of as a recipe. Imagine we are making marinaded steak: we cannot fry the beef before it has been marinaded, and we cannot marinade the beef before we have made the marinade; however, you can make the marinade while I go and buy the beef.

Single instruction multiple data (*SIMD*) or vector processing applies the same processing to many pieces of data simultaneously by replicating very specific parts of the processor. To make another cooking analogy, a particular vector processing instruction can be thought of as a toaster with eight slots. If we want to make toast for four people (two slices each) then it is four times faster than a two-slot toaster. However, if I am just making toast for myself then there is no benefit (in fact we waste power and space on the unused slots). Furthermore as the vector instruction applies the same treatment in a single step it requires us to all arrive at the toaster at the same time with the same requirements. Or it requires some complex engineering so as to work out how we can toast a crumpet on high and a pitta bread on low at the same time. The overhead associated with this coordination limits performance gains. That said, where the same processing is to be applied to a large number of data items simultaneously vector processing is ideal. It is commonly used in computer graphics (processing many pixels simultaneously) and is the basic architecture of most modern graphics processors [194, 29].

Multiple instruction multiple data (*MIMD*) processing replicates enough of the processor's components (in units we call cores) such that it can process multiple different instructions on multiple different pieces of data simultaneously, or give the impression of doing so. Taking the earlier toaster example, instead of having one eight-slot toaster, we now have four two-slot toasters. Each of these toasters can be toasting bread at different settings. This multiplication of computational capacity adds the need to appropriately divide tasks and coordinate between them. While this problem can be tackled in hardware, particularly in how cores should be connected together, the majority of the task of achieving increased performance is shifted to software.

While multiple processor systems have been part of large-scale computers such as those used as servers (providing facilities to many users) or in scientific applications, only relatively recently have multi-core processors become commonplace in commodity computers. The history of research on software engineering techniques to make use of multiple processors and by extension multi-core systems is long; however, the commonly adopted practices in commodity software tools are typically ineffective in both application and implementation (see 2.2.1).

2.1.3.3 Processor Development Trends

It is worth discussing the widely cited article "The Free Lunch Is Over" by Herb Sutter [242]. Sutter's seminal article was published shortly after Intel and AMD, the largest commodity computer processor vendors, announced their respective moves to producing multicore processors rather than attempting to increase processor clock speed. He demonstrates the motivation for a shift to hardware parallelism by presenting data plotting the transistor count, clock speed, power consumption and performance per clock of processors from 1970 to 2005. His data (updated in 2009) shows a clear plateau by

2003 in all measures except transistor count. Figure 3 shows a smaller data set[3] comparing some of the same measures as Sutter. The broad trends match those of Sutter's findings: by the early 2000s clock speed and power consumption plateau, but transistor count continues to grow exponentially.

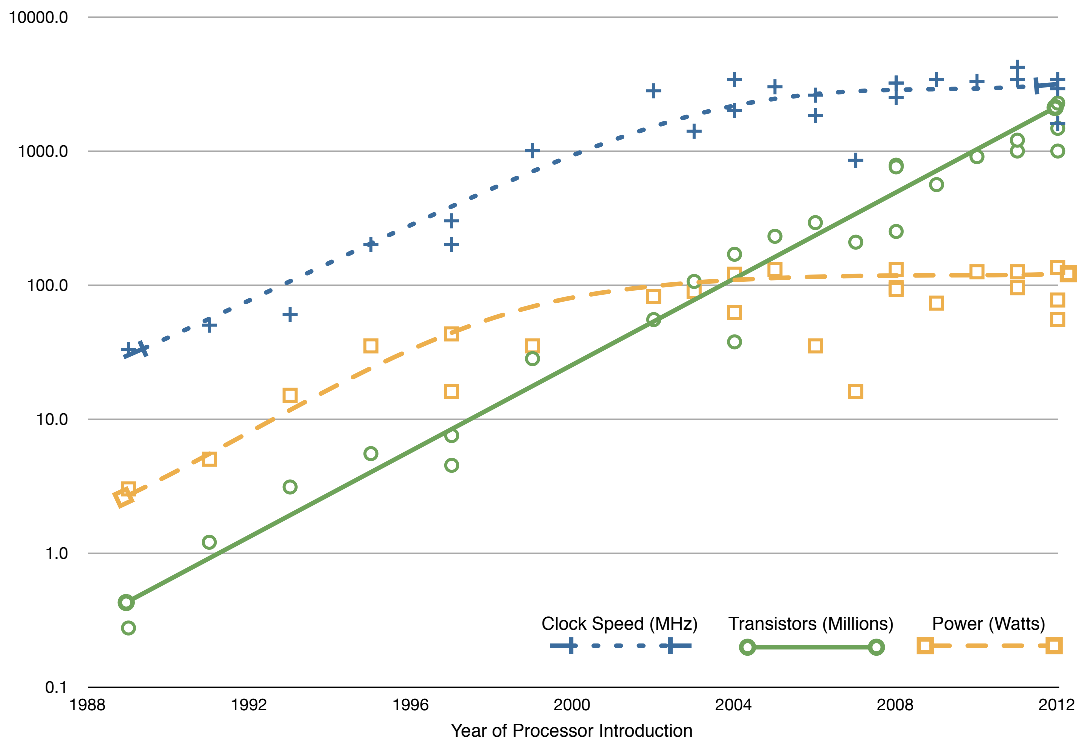


Figure 3: Trends in CPU clock speed, transistor count and power consumption.

Clock speed is the rate at which the clock changes inside the processor. Every clock cycle the next electrical signal (on or off) passes between connected components. It is the fundamental rate at which the electronic components do work. The clock speed of a processor is bounded by power input. This is in turn bounded by the amount of heat a given area of silicon can dissipate. Infamously, one generation of Intel Pentium 4 processors had a thermal dissipation per cm^2 close to that of a nuclear reactor.

[3]Derived directly from manufacturer data sheets.

Performance per clock cycle measures how many instructions are executed per clock cycle. Typically less than one instruction is executed per clock cycle, as it may take a component within the processor many cycles to compute its result. However using *superscalar* techniques such as pipelining and branch prediction, it is possible to raise this number to around five instructions per clock. These techniques require large amounts of additional transistors within the processor and are ultimately bounded by the dependencies that exist within any linear sequence of instructions.

Transistor count within processors continues to increase because the physical limits on transistor size have not yet been reached. This means that with successive generations of manufacturing technology smaller and smaller transistors can be built, which in turn means more transistors can fit in the same physical area. Smaller transistors also require less power to operate. Hence for a fixed area of silicon and a fixed power consumption, the number of transistors can continue to increase (Moore's Law [190]).

Given that the clock speed limit has been reached and the performance per clock is bounded, the only way to use the increasing number of transistors (and yield a significant performance benefit) is to duplicate large elements of the processor. This duplication creates multiple independent cores of computation within the processor and hence multi-core processors are born.

Sutter argues that programmers have been depending on progressive performance improvements between processor generations to make their software run faster. Up until 2003 computer programs would run 50% to 100% faster when the next generation of processor was released. This performance benefit is gained without any change to the program itself, hence the so called "free lunch". With the shift to multi-core this performance benefit is gone. Sutter concludes that programmers will need to rewrite their software and adopt radically different design approaches in order to increase software performance. Specially, programmers will need to use concurrency.

Going beyond Sutter's observations which are strongly rooted in the limits of manufacturing technology, there are other pressures which motivate a shift to multi-core processors.

An interesting parallel can be drawn with the automotive industry where due to instabilities in oil supply, rises in fuel prices and regulatory pressures, a new emphasis on efficiency and alternate fuel sources has become the norm. It could be argued that without these external pressures much less emphasis would have been given to the development of more efficient engine technology. Looking ahead it is possible to see similar pressures influencing the computer hardware industry.

The global population is growing and is expected to continue doing so for the foreseeable future [247]; this leads to significant pressure on resources. Energy is one resource with growing consumption, but also has a shrinking supply due to pressures to phase out fossil fuels and nuclear energy. It is quite possible that global pressure to reduce energy consumption will further engender a shift to multi-core and multi-processor computer systems, based on the assumption that they are more efficient in terms of computation performed per unit of energy consumed. We already see this pressure in two sectors: computer data centres and mobile devices [154, 54]. In computer data centres small changes multiply to create large savings, for example reducing the power consumption of 10,000 servers by just 1W saves 10kW of power resulting in an annual saving of £5,000 [89].

In mobile devices battery capacity is limited and increasingly only slowly, hence efficiency savings increase the attractiveness of devices to consumers. Consumers want the next latest mobile device to be faster, but have the same or better battery life. A device which was twice as fast, but had half the battery life would be a product doomed to failure.

Aside from energy, the resources used to make computers are under pressure or are otherwise at risk. Copper, one of the most commonly used conductive metals has an uncertain future supply, giving it potential price volatility similar to that of oil. The supply of rare earth elements, commonly used in semiconductor production and computer batteries is dominated by China, creating a scenario similar to OPEC's control of global oil supplies (and prices). These forthcoming pressures on resources will favour computer parts which are less resource intensive to produce. I suggest that a multi-core facilitated reduction in processor clock speed can reduce the complexity of manufacturing a given processor, and the complexity of support electronics, particularly if some of the cores take on specialised functions [180].

2.2 Paradigms

Programming paradigms are generalisations of concepts from one or more programming languages. They are not typically formal definitions, but rather styles of implementing computer programs. Different programming languages support different paradigms. It is possible to program any paradigm in any Turing complete programming language; however, the size and hence complexity of the resulting program code will be greatly increased if a style of programming does not map well to those supported by the programming language used. This is similar to how natural languages possess strengths and weaknesses in the expression of certain topics, biases developed from the needs of the native speakers. For example many Asian languages such as Japanese and Korean possess a range of pronouns and associated honorifics which allow the concise and precise expression of the social relationship between the speaker, listener and third parties. This is necessitated by the social norms of these countries; however, expressing the same relationships in English would be difficult without seeming overly verbose.

The fine details of what any given programming paradigm involves are often disputed, object orientation is one such example [149]. This section defines typical concurrent programming paradigms for clarity and later reference. We consider paradigm to mean pattern, and therefore concurrent programming paradigm to mean a set of techniques for achieving and managing concurrency.

Concurrency in computer programming is the potential for one or more elements of a program to be executing simulanteously. To give a generalised example, a computer may be reading input from the user, calculating a result and drawing some graphics to the display at the same time. As described, the tasks occur in parallel; however, depending on the computer's hardware, the programmer's implementation may in fact be executing them in a tight looped sequence such as to simulate parallel operation. When programming this example in a concurrent programming paradigm, each of the three tasks would be written separately. The final program would combine these tasks and in doing so implicitly or explicitly specify how they interact. The programming language's compiler and runtime system would then execute the tasks in parallel or sequentially depending on the hardware support (and required interactions between them). In summary, a concurrent programming paradigm allows the programmer to explicitly admit the concurrent execution of parts of a program. *Concurrency is a gateway to parallelisation.*

We focus on paradigms for structuring computation within a single (mostly homogeneous) computer system, not a distributed environment, although many of the paradigms are extensible to such an environment. Observationally there are two main concurrent programming paradigms, data-oriented concurrency and message-passing concurrency. The following subsections 2.2.1 and 2.2.2 explore these by discussing implementation exemplars.

2.2.1 Data-oriented

In a data-oriented concurrent program the elements executing concurrently interact over one or more units of shared data (shared memory). Typically one or more synchronisation mechanisms are provided allowing the programmer to structure interactions over the shared data.

In a simplistic example a program might split into two concurrent *threads* of execution with each taking half the work. A synchronisation mechanism would permit the two threads to merge back down once the work has been completed.

Minimal work is required to add some level of data-oriented concurrency to existing (non-concurrent) software code. Synchronisation mechanisms such as *locks* (2.3.2.1) are added to data structures to be shared among concurrent components. When one component is modifying a shared resource it locks it so that other components do not interfere (*mutual exclusion*), this is described as taking or acquiring the lock. This methodology is typified by POSIX threads and their associated operations (2.5.24).

Locking, or some form of mutual exclusion, is required to maintain the relative ordering of operations on shared resources (2.3.2.1). To give a contrived example, consider two chefs attempting to cook food in a single kitchen with no regard for each other. One might pick up the flour and move it, leaving the other unable to find it. One might place a pan on the hob, only for the other to turn it off. One might put chillies in the other's chicken soup. Without close coordination the list of possible problems that can arise is seemingly endless. If the two chefs cannot work together then we must only allow one in the kitchen at a time by placing a lock on the door. This is in essence what we are doing when adding locks to shared resources in a computer program. Whether the lock is on the kitchen door, or whether there are separate locks on each of the items in the kitchen is referred to as the granularity of locking, with the former being course-grain locking, and the latter being fine-grain locking.

The apparent simplicity of adding mutual exclusive locking belies the (often subtle) complexity it introduces. A degree of overhead is placed on the programmer: they must remember to acquire the appropriate lock when modifying shared data, and also remember to release it. Forgetting to release a lock will, in all likelihood, result in a program that ceases to function, as concurrent components attempt wait for a lock which is never to be released. This is a condition called *deadlock*.

Locking can be subsumed in to the programming language such that the programmer does not need to worry about it or not consider it explicitly; Java's `synchronized` [117] keyword is one example of this (2.5.17). However another danger quickly arises, assuming I have two locks A and B and two concurrent components X and Y, where component X locks A then B, and component Y locks B then A. If the execution of these components is interleaved such that X locks A, then Y locks B, both components will deadlock attempting to obtain their second lock. This scenario can be avoid by placing an order on locks, such that B can never be taken before A (changing the behaviour of component Y). Critically such an ordering cannot easily be applied by a programming language or its support systems, and can place a significant burden on the programmer.

Given the stated complexity of multiple locks it would seem logical to have fewer locks. However, using fewer locks means that concurrent components spend more time competing or waiting on shared data, although there is reduced locking overhead. Although performance may not be the primary reason for a use of concurrency (2.1), increased competition for fewer locks or from more concurrent components sharing the same data is detrimental to performance. Take the extreme example of two components which share data via a single lock. If both components spend all their time manipulating shared data, but only one can be active at a time due to the lock, then the performance is theoretically the same as having a single component with no lock. In practice performance is worse than a single component without a lock due to the overheads of the lock, time spent switching between components and other hardware concerns. In fact the reality is that performance for naive decompositions of existing non-concurrent code may not show significant improvement [45], unless they contain what are described as embarrassingly parallel problems (2.1.2). Embarrassingly parallel problems are those that are inherently decomposable by their nature, thus leading to concurrent solutions with minimal or no shared data.

2.2.1.1 Vectorization

When dealing with computations which are inherently data-oriented, but admit an obvious decomposition, such as the Mandelbrot set or processing image elements, then vectorization can be applied. Simplistically vectorization takes program code required to process a single unit of data and replicates it across all available processing resources. This might use single instruction multiple data instruction level parallelism as mentioned in section 2.1.2. For example if we need to add one to 1000 numbers in an array and our processor has four arithmetic units, vectorization could give close to a four times speed up (bounded by data transfer restrictions). Vectorization might also use multiple processor cores, each of which might be internally parallel. Critically vectorization is a very specific form of concurrency where all concurrent components are executing the same code in parallel on separate pieces of data so as to eschew the need for synchronisation and locking.

Intel's Thread Building Blocks (2.5.16) is an example of vectorization on modern multi-core processors. Nvidia's CUDA (2.5.14) provides similar functionality for multi-core graphics processors which have a very high degree of instruction level parallelism. OpenCL (2.5.14) combines both by providing programming language extensions for constructing parallel code on both modern multi-core and specialised processors such as graphics processors.

On a theoretical level vectorization techniques can be seen as a special case of Bulk Synchronous Processing (BSP) [248]. In BSP a number of independent processors perform computation while exchanging messages to update or access non-local data. At regular intervals all processors synchronise state, processors which reach a synchronisation point wait for all others to reach the same point. Theoretically BSP has optimal utilisation of computation resources for applicable algorithms. With respect to vectorization, the synchronisation of state occurs when all parallel operations complete and the program is running sequentially before more parallel operations are initiated. The implication is that vectorization has near optimal utilisation of computation resources. A major caveat is that vectorization does not provide independent processors. Parallel computations must be in step at an instruction level to gain optimal performance.

2.2.1.2 Transactional Memory

Transactional memory takes inspiration from database transactions to alleviate synchronisation issues such as deadlock when using locks in data-oriented concurrency [122]. Instead of locking shared data, updating it and releasing the lock, transactional memory systems allow the programmer to declare a group of operations which occur as an *atomic* transaction. A transaction and its results either occur in full and are made visible to other concurrent components together, or are not observed to have occurred at all.

To take a simple example, let us look at incrementing a number. There are three operations involved:

1. Read the present value from memory
2. Calculate the new value by adding one
3. Store the new value back to memory

If two concurrent components execute these operations at the same time such that they interleave then from a starting value of n then the possible output states are $n + 1$ and $n + 2$. Obviously $n + 1$ is not the expected result of incrementing a number twice. With transactional memory, these three operations would be grouped together as a transaction guaranteeing that the memory written to in step 3 had not changed since step 1 completed.

Most computer hardware supports small transactions similar to the above on machine words, which are typically four or eight bytes in size [143]. The relative complexity and cost of implementing large transactions in hardware has kept transactional memory out of all but the most specialised processors, for example Sun's Rock processor [253], Azul's Vega architecture [73] and IBM's Blue Gene/Q [121]. As computer programs work on much larger pieces of data than a few bytes the majority of research on transactional memory has involved software transactional memory (STM) [232].

The transactional memory approach is optimistic as arbitration only needs to occur when two concurrent components conflict, for example manipulating the same data at the same time. Locking itself can be seen as pessimistic as the cost of taking the lock is always paid regardless of whether there is contention to access the shared data or not. That said, arbitration of arbitrary size transaction is complex and potentially costly. While there are many different techniques most involve locking memory at a word or region level for some or all changes. This also makes software transactional memory more suited to managed runtime environment languages such as Java, where all runtime activity is at some level arbitrated by the language runtime. It is also important to note that STM itself does not provide any means for synchronising threads.

In points of the program where synchronisation is required (e.g. producer with consumer) external means must be used in order to avoid naive polling. This means that transactional memory is not a replacement for more general concurrency, but is another tool for data-oriented concurrency.

While simplifying the programming model required to safely use and manipulate shared data between components, STM has a relatively high cost (in machine resources). Critically these costs may not always be amortised by the increase in concurrency gained, particularly when compared to an efficient fine-grain locking solutions [104]. To realise the potential of transactional memory, it appears that hardware support is required. Intel have announced Transactional Synchronization Extensions for their future processors [142] a restricted form of transactional memory, suggesting transactional memory will soon see more widespread applicability. However, Intel's announcement should be counterpointed by the observation that the number of mobile devices embedding non-Intel processors, but possessing multiple cores, far outstrips the deployment of Intel processors.

2.2.1.3 Partitioned Global Address Space

A number of parallel programming languages have adopted a programming model based on a *partitioned global address space* (*PGAS*) [76]. Data-oriented programming gives the impression that all memory is equally accessible from all points of the system. In large computer systems this is not true, even if all memory is accessible from any point in the system the performance of memory will vary based on its distance to the processor (2.6). Typically each computational element will have an area of memory which is local and has better performance. A PGAS model acknowledges these differences by allowing the placement of computations within partitions so they may use fast local memory and minimise communication with remote memory. Many languages designed for parallel computation on massive computer systems use a PGAS model, examples include Chapel (2.5.7), X10 (2.5.29) and Unified Parallel C (2.5.28).

2.2.2 Message-passing

In a message-passing program components executing concurrently interact by sending messages to each other. Program components are isolated and do not share any memory, data is shared by sending it (by value) in a message from one component to another.

To mirror the example in section 2.2.1, a program component would split the work into two separate messages. Each message, containing half the work, would be sent to a worker component. When a worker finishes it would send a message back containing the completed work.

Message-passing is inherently concurrent due to the way it structures program components. This contrasts with data-oriented concurrency which is for the most part an extension of sequential programming techniques. For this reason, data-oriented concurrency is often favoured, because it would appear to require less restructuring when applied to existing program code. However by making concurrency an explicit part of program structure, message-passing concurrency removes or mitigates many of the pitfalls associated with data-oriented concurrency, such as the need for locks on shared data. There is also an obvious mapping between a message-passing software design and distributed physical components, for example separate computers connected via a communication network. We can see how processes could be distributed between the computers and messages passed via the network, but it is less clear how a data-oriented design would be mapped to the same structure.

Programming languages and systems for message-passing concurrency have been heavily influenced by mathematical models of concurrent computation. These models can be broadly divided into two main formalisms:

- The Actor model [130, 74, 25],
- Process calculi (e.g. CSP [134, 226], CCS [183], the pi-calculus [185, 184] and join-calculus [111]).

These various computational models share much in common, in part due to a cross-pollination of ideas between their creators. The most significant difference is the Actor model's use of addressed messages conveyed between processes by a medium with no defined properties, which contrasts with the use of well-defined communication channels to carry messages within the process calculi. Also of note is the variance between process calculi with respect to whether communication is asynchronous or not; Actor model communication is asynchronous. These two differences can be analogised as the difference between posting a letter and making a telephone call. Asynchronous message-passing is like putting a letter in the post with an address on it: we cannot make any assumptions about when (or even if) the destination will receive the letter. In fact I could send two letters on two successive days, and they might arrive in the opposite order. Synchronous communication is like making a telephone call, after dialling the number I wait, when the other party picks up I can convey my message then put down the phone safe in the knowledge that my message has reached its destination (assuming the other party was listening). This synchronous communication makes most sense with channels, which could be likened to connected phone lines. Once I've sent my first message I leave the line connected; when I come to send another message I simply speak again and wait for an acknowledgement from the other end. The channel guarantees the order of these two messages (unlike the two letters).

Pragmatically it is possible to argue that the Actor model and process calculi represent broadly the same functionality. Synchronous communication and channels can be implemented on top of an asynchronous communication medium as demonstrated by the Transmission Control Protocol (TCP) which underpins the modern Internet and World Wide Web [243]. However this comes with considerable associated complexity in order to manage reordering and loss of data, multiplexing and timing issues, as attested to by TCP's 85-page specification, numerous supporting documents and widely documented implementation bugs. Conversely, implementing asynchronous communication with synchronous channels can be achieved by providing all processes with a channel to a *mailbox* process, these mailboxes are in turn connected to a *post office*.

When a process wants to send a message it puts it in its mailbox, which in turn sends it to the post office for appropriate routing. To receive new messages the process simply queries its mailbox. The obvious contrast between these two examples derives from the fact that it is simpler to weaken the strong guarantees provided by synchronous communication and channels, than it is to add guarantees to asynchronous communication.

In terms of implementations we see Actor model based concurrency implemented in many actively developed programming languages; notable examples include: *Erlang* (2.5.11), *Clojure* (2.5.9) and *Scala* (2.5.26). The Actor model itself and these implementations have been heavily influenced by *Smalltalk* (2.5.27). Smalltalk's influence can be seen in mainstream languages, such as *Objective-C* (2.5.20), where messages are sent to objects to invoke methods. Process calculi inspired concurrency has had less obvious influence on modern programming languages; however, notable examples are: *Ada* (2.5.2), *Concurrent ML* (2.5.10), *Go* (2.5.13), *occam* (2.5.21) and *XC* (2.5.30). Historically we see *CSP* [226] having the most influence on modern programming languages, largely through the work of Bell Labs on the Unix operating system [216], *Plan 9* [203] and languages such as *Newsqueak* (2.5.13). Within the super-computing and high-performance computing communities we see use of *Message Passing Interface* (2.5.19) to support portable message passing between different systems and languages.

2.2.3 Other Paradigms

Having discussed the two main concurrent programming paradigms it is important to relate these to specific cases of concurrency in other paradigms.

In *event-driven programming* components are defined to handle (be activated on) external or internal events. This technique is commonly used in user interfaces where the program is principally driven by events from the user such as mouse clicks. Events can occur concurrently, but might be queued and handled sequentially, particularly if they share data. If components can be executing concurrently and data is shared, then data-oriented concurrent programming techniques will need to be applied. Alternatively event-driven programming can be accurately modelled by message-passing, where events become messages to concurrent components which handle them.

Functional programming expresses program components as mathematical functions. In a pure functional programming language functions have no side-effects or state; they deterministically compute a result from their input parameters. This means that functions do not share state and hence many of the pitfalls of data-oriented concurrency are avoided. Additionally, as a function's output is wholly determined by its input, thus it is possible to use functions themselves as transactions in software transactional memory systems (2.2.1.2). In a lazy functional language computation of function results is deferred until the results are required; alternatively these results could be computed concurrently in what is often referred to as a *promise* or *future* (2.3.2.5). This is a form of automatic parallelisation and does not in itself provide an expression of concurrency. Furthermore as functions are typically small it is not efficient to execute them all concurrently, therefore the complexity of functions must be computed and annotations added to decide direct concurrent execution [135]. Allowing the programmer to manage the execution order of concurrent functions is a key component of the Cilk programming language (2.5.8).

Stream programming is a limited form of message-passing where data *streams* move through a static network of concurrent components [116]. This structure is effective for digital signal processing applications such as image, video or sound data. Data messages contain very small pieces of data such as image pixels and are buffered between concurrent components. This allows concurrent components to be scheduled so that multiple instances of the same component are executing in parallel using vectorising hardware such as graphic processors [62]. The primary function of stream programming systems is the efficient automatic placement and scheduling of concurrent components to achieve maximum throughput this is aided by the use of static networks [156]. As such stream programming is specialised to parallel programming and asymmetrically concurrent or reactive systems.

Coordination or generative languages such as *LINDA* provide a hybrid between data-oriented and message-passing approaches [113, 114]. In LINDA specifically, data is shared between concurrent components in a globally accessible and persistent store called a *tuple space*. Concurrent components read from and write data (*tuples*) to the tuple space. Importantly, reads to non-existent tuples wait for the tuple to become available and on completion a read removes the tuple from the tuple space (unless directed otherwise). Hence while the shared store is data-oriented it provides explicit communication. In a sense the tuple space is an array of buffered communication *channels* (2.3.2.4). Coordination languages draw heavily on the CSP process calculus, and the *Ease* coordination language has been referred to as *process-oriented* (2.4) [102].

2.3 Primitives

There are many different terms in use for what can be seen as broadly similar concepts used in the programming of concurrent computer software. For example the terms process, thread, actor and agent are all used (with varying semantics) to mean concurrently executing parts of a computer program. This section describes common primitives available in programming languages. Specifically it explores unifying names and definitions of each for use within this thesis.

2.3.1 Processes

Processes are the concurrent unit of execution in process-oriented programming. The use of the word *process* originates in multiprogramming systems of the 1960s and defines an executing computer program and its current state (or context). This disambiguation, with respect to program, is required as there may be multiple running instances of a given program each with their own state. Thus the prevailing understanding of the term process is that of an operating system defined construct for managing a given execution of a program.

The operating system mediates interactions between processes and enforces boundaries; for example processes may not be able to write to the memory space of other processes. To the operating system there is one process per execution of a program. While most operating systems allow programs to create additional processes, each of these is considered a separate running program with its own protected state.

Many names are used for concurrent units of work with varying semantics. As noted above the term *process* refers almost universally to isolated units of concurrent computation (separate program invocations). In most programming languages these are considered operating system managed primitives. This definition is generally compatible with the definition of processes in process calculi (2.2.2).

2.3.1.1 Threads and Fibers

The most commonly mentioned unit of concurrent computation is the *thread*. Threads are an operating system concept with similar origins to processes; however, whereas processes are isolated program instances, threads are concurrent elements of the same program. From an operating system standpoint, threads divide a process into separate concurrent units of execution all of which operate within the process's memory and state. Threads may be managed by the operating system, *kernel threads*, giving them a similar level of support to processes; these threads are essentially light-weight processes. This model is typified by POSIX threads (2.5.24). Alternatively threads may be managed by the process within which they run, *user threads*: this can be done without operating system support as in GNU Portable Threads [6] and Java Green Threads [198].

There is a general understanding that threads are pre-emptively scheduled. When there are more threads than processor cores to execute concurrent elements in parallel, threads must be swapped in and out of processor cores. In a pre-emptive environment this can happen at any point. For example when a thread has spent a certain amount of time executing it is swapped in order to give another thread some time to execute, thus allowing all threads to make progress. Imagine I have two pans of soup to warm up, but only one hob, I can swap the pans at regular intervals so they get an equal share of the heat. I am pre-empting the heating of one pan whenever I swap to another. I could also pre-empt both and cook some bacon on the hob. The significant danger in this example is that if I try to divide time between too many pans I will make no progress cooking any food [4].

The alternative to pre-emptive scheduling is cooperative scheduling. In cooperative scheduling components are only swapped out when they release the processor. For example a component might do as much work as it can then release the processor so other components can produce more data for it to work on. Cooperative scheduling is commonly used with event-driven programming (2.2.3) where components that process events release the processor after they finish handling a given event. Cooperatively scheduled light-weight processes are sometimes called *fibers* [193]. However, scheduling concerns aside there is no difference between threads and fibers except a perception that the latter require fewer resources to implement.

2.3.1.2 Actors, Objects and Agents

In discussing message-passing concurrency we mentioned *actors* and the actor model (2.2.2) [130]. The actor model has well-defined semantics [130, 25, 74]; however, the behaviour an actor can possess has very few restrictions. Essentially actors are expected to capture the behaviour of all separate system components. For example the system scheduler itself may be an actor, or the processor an actor which executes commands on behalf of other actors. The only required behaviour of actors is that they communicate using asynchronous messages sent by address. In practice this means that all other types of process, thread or fiber can be called actors.

In object-oriented programming everything in the world or system is modelled as an *object*. Objects have methods which can be activated to produce some results, for example a calculator object would have a method for each button on the calculator.

[4]While the results of a computer program do not become cold as such they may only be useful if produced within a given time window. Cache must also be kept *warm (up to date) to be effective (2.6).*

Subsequent button presses alter the internal state of the calculator object, state which we can interrogate by perhaps calling a method to read the screen. Objects belong to one or more classes and the power of object-orientation is that code written to manipulate one object can in principle manipulate all objects of the same class. Real world objects are inherently concurrent; however, common object-oriented programming systems such as Java (2.5.17) and C++ (2.5.6) do not provide objects with concurrency. Thus the notion that I can turn on the kettle object and toaster object together to make my breakfast can be hard to implement in an object-oriented language.

Another term used in a similar manner as actor to describe concurrent components is *agent*. Agent-oriented programming has a very specific definition as a specialization of object-oriented programming [234]. However in studying Shoham's definition of agent-oriented programming it can be seen that it was intended as a realisation of Hewitt's Actor model [130]. Despite this the term agent is used broadly due to interdisciplinary understanding of the word's etymology, the Latin agere "to do". Thus *agent* is used to refer to autonomous entities in computer simulations, which are in essence actors without the inherent semantics of the Actor model.

2.3.1.3 Closures, Coroutines and Continuations

A *closure* is a *function* together with the environment it references. The function can be thought as a recipe which describes how to take a set of ingredients (parameters) and produce a result. In this case a closure is the function with some of the ingredients supplied. For example, I buy a loaf of bread with a recipe for making sandwiches which reads:

1. Cut two slices of bread
2. Insert filling between slices

Since I have the bread (supplied with the recipe) I can replace "bread" with "this loaf of bread". This is now a closure for making sandwiches. Now each time I use this recipe (execute the closure) I add some filling of my choice (supply parameters) and a sandwich is produced as a result. The *side effect* is that the loaf of bread is made smaller each time I use the recipe.

Closures are an implicit part of functional programming languages such as ML [186], Scheme [100] and Haskell (2.5.15), but closure like facilities are also present in other high-level programming languages. Objective-C provides *blocks* which are for all intents and purposes closures (2.5.20). This is of interest as Objective-C uses closures to provide concurrency within sequential program code. The programmer can create a block at any point and then schedule it for execution in response to an event, or place it on a job queue where it may begin executing concurrently to the program component which created it. This model can be managed by a scheduler which simply executes closures on available processors in the order they arrive on the job queue. In fact this is just cooperative scheduling and the closures are effectively fibers (2.3.1.1) which release the processor when they finish. Thus the programmer must still address data-oriented concurrency concerns.

In some contexts functions are referred to as *subroutines*. This definition originates in the description of a computer program as a routine or exercise the computer hardware performs. Historically the first computers would perform only a handful of routine tasks, such as calculating ordinance trajectories or payroll [106]. To help with structure, construction and understanding a routine (computer program) was broken down into or composed from subroutines. A program executes by calling subroutines to do work, those subroutines might in turn call further subroutines, and so on. Each subroutine has a distinct start and end, although unlike a pure function there is no implication that a subroutine takes any input or produces an output. Formally this can be viewed as the subroutine taking the entire state of the program as its input and producing a new program state as its output.

Built on the subroutine concept are *coroutines* [82]. Coroutines are a superset or generalisation of subroutines. While subroutines have a single start and end, coroutines can stop by calling (yielding to) other coroutines and when later called again continue executing at the point where they left off. This is useful where one subroutine consumes data produced by another. The producer can call the consumer when data is ready and the consumer can call the producer when it needs more data. While the coroutines are not executed concurrently, they are active concurrently with respect to their state. Again, this is a restricted example of cooperative scheduling.

An important concept for implementing coroutines and similar programming concepts is the *continuation* [251, 125]. A continuation captures the running state of the program (or program component). Unlike a closure which can still have parameters to fill in and has not yet started, a continuation represents a point of execution. If I am baking cookies to a recipe and have to leave the kitchen to answer the telephone, I remember where I have reached in the recipe. The point I have remembered is the continuation. When I come back to the kitchen, I continue where I have left off; however, the state of the kitchen is not part of the continuation, so someone could have been in and eaten the chocolate chips while I was out of the room. I could however hide the chocolate chips before leaving the kitchen, and likewise program components can be designed to establish similarly "safe" states before creating a continuation.

Closures, coroutines and continuations model concurrency at the programming language level whereas processes, threads and fibers express concurrency in terms of units managed by the operating system or another component such as a language run-time system. A concurrent programming model is achieved by establishing a relationship between these components. In the extreme one class of components is made implicit by abstraction. For example in pure functional programming, processes and threads are not expressed at all, but simply a feature of implementation. Conversely in an imperative programming language, closures, coroutines and continuations are not expressed, but rather are implicit features of how the program is prepared for execution.

2.3.1.4 Scheduling and Priority

I have described *pre-emptive* and *cooperative* scheduling in relationship to the execution of concurrent program components. Scheduling is necessary as there are rarely sufficient resources to have all concurrent components executing in parallel, or necessarily a need for them to do so. Unless a computer program is designed to function directly on the target computer system, even the most basic program expecting to run exclusively, *single-tasking*, on a given computer system will still need to share processor time and other resources with the operating system. In this simplest case the program invokes the operating system to perform tasks and the operating system returns to the program when finished, a form of cooperative scheduling. Pre-emptive scheduling was born of two principle concerns:

- the operating system may need to *interrupt* the running program to handle an urgent event,
- by dividing processor time between multiple tasks a computer system can be shared between separate tasks.

The second of these originates in the relative high cost of computer systems, necessitating sharing to maintain utilisation. This was particularly true as computers became interactive rather than simply job based (where the user would submit jobs to an operator for later execution). Interactive computer usage produces spikes of utilisation as the computer is fundamentally idle between user commands, thus multiplexing the system between multiple users is both possible and desirable to avoid wasted processor time. Such interactive sessions could also be multiplexed with non-interactive and potentially long running jobs. A full discussion of these concerns and their implications for operating system design can be found in operating system texts such as those by Tanenbaum [244].

Having established the need for, or otherwise presence of, scheduling, with it we gain the ability to choose which component runs at a given time. Control is typically exercised in two ways:

1. when one component finishes or otherwise yields the processor a new component is selected,
2. the running component could be pre-empted and another component allowed to run in its place.

Given sufficient a priori knowledge of the behaviour of (all) components an execution plan could be computed to meet goals such giving all components equal processor time. However, the behaviour of components is seldom known in advance particularly in the presence of unpredictable events such as user input. This means that in practice heuristics are used to approximate the results of an execution plan. One method used for the application of these heuristics is *priority*.

Each concurrent component is assigned a priority by design (*static priority*), by a scheduler, or by some combination of both. When selecting the next component to run, the highest priority component is selected. If a component is created or otherwise becomes active, perhaps as the result of an external event, and the newly activated component has a higher priority than the currently running component, then the currently running component may be pre-empted. An extensive range of scheduling behaviours can be implemented by modifying the priority of a component in response to changing conditions (*dynamic priority*). For example a long running component could have its priority reduced so that other components with more short-term needs are serviced first. I will leave a more detailed discussion of such scheduling heuristics to Chapter 3 where it is most relevant. There are however some related concepts which I elucidate further here.

If priority is strictly observed and high priority components are always available to run then lower priority components may never acquire any processor time. This is called *starvation*. A heuristic for avoiding starvation is to periodically increase the priority of components which have not received any processor time so they will eventually gain sufficient priority to run. The priority must then be reset. This heuristic can have undesirable consequences if the designer assumed that low priority component would never run while high priority components are active.

Another unintended circumstance that can occur with priority and shared resources is *priority inversion*. During priority inversion a lower priority component is able to execute even though a higher priority component is available. Assume three components H, M and L with respectively high, medium and low priorities. Component L claims a shared resource, then due to an external event component H becomes active. H is of higher priority than L so L is pre-empted. Now if H tries to claim the shared resource, but must wait because it is held by L. H yields so that L becomes active again. Now if M becomes active either by external event or as created by L then L will be pre-empted. Assuming M does not interact with the shared resource, M has achieved priority higher than H. If M starves L, then H will also starve. This example highlights the impact shared resources can have and their performance implications with respect to Amdahl's law [30].

Amdahl's law is a model postulated by Gene Amdahl to describe the relationship of the parallel speedup of an algorithm in relation to its sequential version [30]. It relates speedup to the proportions of the algorithm which are sequential and parallel. Expressed simply, an algorithm can never go faster than its slowest sequential component. The sequential portions of an algorithm place an upper bound on the speedup that can be achieve regardless of the available parallel computational resources. This can also be applied to hardware resources such as a single shared memory, which is a single sequential component.

It is important to note that the Actor model and most process calculi such as pi-calculus and the join-calculus do not consider priority with respect to concurrent components. This is not to say that priority is precluded, but rather that there is no inherent mechanism for expressing it and hence reasoning about its impact on a component's behaviour. Much work has been done on extending process calculi with notions of time and priority, where the former is often required for the latter [72, 107, 87, 226, 188]. The comparatively weak semantic guarantees of the Actor model are unaffected by the presence or absence of priority. In practice most programming languages that support concurrency based on or inspired by the Actor model and process calculi *do* provide a means of implementing priority either globally on concurrent components or locally through choice over concurrent events (see 2.3.3). As such language features are engineered in lieu of mathematical models, mathematical expression of these could be considered post hoc capture of existing behaviours [225, 65].

2.3.1.5 Performance

It is important to mention the performance implications of the primitives so far outlined. Essentially each of the primitives has an associated overhead in memory space and processor time. These overheads can be quantified as *context size* and time complexity of a *context switch*.

In a scheduled environment where concurrent components share processor cores, a *context switch* is the passing of control of a processor from one concurrent component to another. When a context switch occurs, the context (running state) of one component must be stored (implicitly or explicitly creating a continuation), then the context of the new component must be restored. For operating system managed components such as processes and threads the context switch overhead is often significantly higher than language based constructs such as as fibers, actors, coroutines and continuations. This is due to a number of hardware related concerns the operating system must manage. For example each operating system managed process has its own map of system memory in the form of *page tables*. These tables isolate processes from each other and provide protection so that one process cannot overwrite memory used by another. At a context switch the operating system must acquire privileges (managed by the computer hardware) to change these tables, then look up and adjust various hardware settings, and finally synchronise the hardware with the new intended memory map. These steps can take upward of 3000ns on present commodity hardware, whereas context switches between concurrent components managed by a programming language runtime may only be 30ns.

The operating system related context switch overhead is high as operating system induced context switches are not typically cooperative. This means the state of the component cannot be assumed and all state must be preserved. Context switches induced by language constructs such as fibers, actors, closures, coroutines and continuations are cooperative, the running component is aware of the switch and thus only required state need be stored. This effect can be further enhanced when the programming language manages state with a priori knowledge of when context switches may occur. At the simplest level a subroutine call could be considered a context switch where in the design of the programming language the state preserved by the call is agreed in advanced. If the caller uses state which it knows not to be preserved by the calling agreement it can save this prior to the call; going further it can be optimised not to use state not preserved by the call inherently reducing the context switch overhead. Conversely if a subroutine does not call any other subroutines it can be optimised to use state that is not preserved.

To give an example of the above cooperative context switching, consider a kitchen shared by two chefs. There is only one chef using the kitchen at a time and when one chef leaves they always cleaned and tidied away the utensils to the same places. This means the next chef does not need to search for or clean the utensils next time they enter the kitchen. Both chefs work to maintain the agreed state, saving both time and effort later on. The operating system manages which chef is using the kitchen at a given time, but does not need to do anything in the kitchen itself. This analogy is faithful to real world scenarios. However, applying pre-emptive scheduling to the kitchen is somewhat unrealistic. With the pre-emptively scheduled kitchen the operating system can come into the kitchen at *any* point and render the chef unconscious by hypnosis or some other means. The chef is then taken out of the kitchen and the operating system carefully records the position and state of all the utensils. If some of them are dirty then those must be kept and swapped for clean ones. When a new chef starts work the kitchen is in a clean known state, but whenever a previously pre-empted chef is returned to the kitchen all of the utensils must be replaced exactly as recorded so that no difference is observed.

The above example highlights the inherent cost of a "heavy-weight" pre-emptive approach. However, the cost can be significantly reduced by adopting a partially cooperative approach. This can be achieved by only allowing pre-emption at defined *safe points*. Such an approach can supported by dedicated hardware such as in the INMOS Transputer processor [138, 189]. I will discuss this further in Chapter 3.

2.3.2 Communication and Synchronisation

Communication is the passing of data or other program state between concurrent components. For the purposes of hierarchy I consider synchronisation to be a form of communication. Synchronisation allows reasoning about the state of two or more concurrent components. For example, if two processes synchronise we know they have both reached a given point in their execution.

2.3.2.1 Semaphores and Monitors

One of the earliest proposed and implemented forms of synchronisation between concurrent components is Dijkstra's *semaphore* [91, 94].

A semaphore contains a count and provides two operations for modifying it:

P (called *wait*) decrements the count,

V (called *signal*) increments the count.

Critically these operations are *atomic*: their inner workings are indivisible such that no interleaving of operations can damage or corrupt the count. Synchronisation is established by maintaining an invariant on the count, which maintains that the count may not become negative. Hence if a *P* operation would reduce the count below zero it *waits* until *signalled* by a *V* operation. In such a scenario the *V* operation effectively cancels the *P* operation.

Semaphores can be used to manage the sharing of a resource by allowing only as many concurrent users of the resource as the initial count of the semaphore. When program component wants to use the resource it decrements the count, if there are no available resources the count will be zero and the user will have to wait for one of the existing users to release the resource and increment the count. Semaphores can also provide mutual exclusion between concurrent components. A lock can be created using a binary semaphore, a semaphore which has an initial count of one. The lock is taken with a *P* operation and released with a *V* operation.

A later but related concept is that of Brinch-Hansen and Hoare's *monitor* [119, 133, 120]. Monitors provide mutual exclusion, but at a higher semantic level than semaphores. While semaphores exist separate from the resource they protect, monitors are implicitly embedded within the resource. A monitor encapsulates data and a set of operations on that data. This is comparable to the concept of an object in the object-oriented style of programming. The monitor ensures mutual exclusion of operations within the monitor. If a program component initiates an operation, any overlapping operations initiated by concurrent components are made to wait until the first completes.

There are times when concurrent operations are required. In particular it may not be possible for an operation to complete until some condition is satisfied by the operations of other concurrent components. Imagine a kitchen controlled by a monitor who manages operations using the kitchen. I initiate the operation of making custard, but there is insufficient milk. Milk is delivered regularly to the kitchen, but because I am using the kitchen a delivery cannot be made. I do not want to cancel the operation of making custard as I've done most of the preparation already, instead I tell the monitor I want to *wait* on the *milk available* condition and go to sleep in the corner of the kitchen. In the mean time others can come and go using the kitchen, including the milkman who can deliver milk. On delivering milk, the milkman *signals* the *milk available* condition and the monitor wakes me up.

An important observation is that semaphores and monitors mirror each other's functionality and in a sense are duals of each other. A monitor can be used to implement a semaphore with relative ease. The count of the semaphore is held in the monitor along with the mutually exclusive P and V operations along with a condition for *greater than zero*. The P operation checks the semaphore count and waits on the *greater than zero* condition if the count is zero. The V operation increments the count and signals the *greater than zero* condition. Conversely a monitor can be implemented using semaphores, but care is required. A binary semaphore can provide the monitor's mutual exclusion of operations. Separate semaphores can be used to implement each condition, these semaphores have an initial count of zero, and P operations used to wait for the condition and *V* operations to signal the condition. Care must be taken that conditions signalled when there are no waiting components do not increment the semaphore, otherwise the semaphore will not cause future components to wait and cease to serve its purpose. Additionally this design does not maintain the strict semantics outlined in Hoare's original design [133].

There is a divergence in monitor behaviour when there are multiple components waiting for the same condition. Hoare's initial design specifies that exactly one of those waiting should be woken up and be given preferential access to the monitor, for example in the above example once I have been woken up by the monitor and the milkman has left the kitchen I am guaranteed to be the next to use the kitchen thus ensuring that I get to use the milk and finish making my custard. From an implementation perspective this implies that there is a queue of components waiting to perform operations, queues of components for each condition and that it is possible to insert items at the beginning of the operations queue so as to guarantee priority for signalled components. The last of these implications is difficult to achieve if a monitor is implemented using semaphores. This is because there is no way to pre-empt the queue of components waiting on a semaphore. Various solutions to this problem have been proposed and implemented [244], most of which involve using multiple semaphores to separate operations woken up by conditions from operations waiting on the monitor itself.

An alternative to increased implementation complexity is to relax Hoare's constraints and place components woken by condition satisfaction on the same queue as components waiting to perform operations. In the previously outlined semaphore implementation this means that having woken from a condition semaphore the component attempts to take the operation binary semaphore again, competing with other components attempting operations. Applying this to the kitchen example, instead of sleeping in the corner of the kitchen, I wait outside the kitchen and when told the condition is satisfied I try the door again, queuing with others waiting for the kitchen. The implication is that despite being told there is milk, three people could have made hot chocolate with it and used it all up by the time I get in to the kitchen again. This means that

I must retest the condition and could wait many times (perhaps forever) for it to be satisfied [262].

A further simplification applied to monitor implementations is to only have a single queue on top of which all other conditions are mapped. When a component needs to wait on a condition it waits on the shared queue. If there is only one possible condition then a single component can be woken from the queue, but if there are multiple possible conditions then all waiting components must be woken. Each component then takes its turn operating on the monitor and testing whether its condition has been satisfied, after which it might finish operating on the monitor if its condition has been satisfied or wait again if not. While this implementation saves memory space and processor time as only one queue need be stored and maintained, it has significant inefficiencies. As the number of conditions or waiting components increases, the time taken to signal them all, and for all of them to be scheduled and test their conditions is increased. While overall time is increased, the amount of time spent doing meaningful work, by the component which has satisfied its conditions, remains constant. This is compounded by the fact that components operating on the monitor simply to test their conditions are excluding other components without any conditions from using the monitor.

Despite the performance implications, simplified monitor implementations are provided in many common programming languages, such as Java (2.5.17) and C# (2.5.5). Simplified monitors are appropriate when there is only one condition and it could be argued that in the structured style of programming that typifies these languages that only one condition should be present in any given monitor. Where multiple conditions are required there will need to be multiple monitors. This does however reduce the semantic benefits of monitors over semaphores. Essentially while semaphores and monitors provide the same functionality (mutual exclusion), when implemented as part of a programming language such as *Concurrent Pascal* [120] monitors provide a tight coupling of this functionality to the operations and data it protected, whereas semaphores are separated from the data protected.

When programming, at least imperatively, it is not unreasonable for the programmer to imagine themselves performing the actions of the program. Essentially the programmer role plays the scenario from the point of view of the computer and writes down a list of the actions they would take, with a level of generalisation. Applying this principle to the previous kitchen examples, I as the programmer am a chef who uses the kitchen. If the kitchen is protected by a semaphore then there is no door to the kitchen, but an indicator next to it showing whether it is available or not. When I go into the kitchen I check and update the indicator, or wait until the kitchen is available. When I leave the kitchen I also update the indicator. I can however forget to consult or update the indicator and wander into the kitchen while it is in use, or leave it without indicating it is now available. Both of these scenarios have significant implications as I am not expecting to coordinate my actions with any other concurrent users. As there is no door I can also peer into the kitchen and look at what is going on. If the kitchen is protected by a monitor then there is a door which automatically locks from the inside when someone is in the kitchen. When I want to use the kitchen I have to pass through the door; I cannot avoid it. If the door is locked I have to wait; I also cannot see inside[5]. Since the lock is automatic I cannot forget to lock the door when I enter, and I have to unlock it to get out again.

In conclusion semaphores and monitors provide synchronisation and mutual exclusion. Monitors are strongly coupled with the data or resources they protect and thus can reduce programmer errors compared to semaphores. When used for mutual exclusion both semaphores and monitors have the same performance implications with respect to Amdahl's law [30]. As locking primitives both also increase program complexity as per data-oriented concurrency (2.2.1).

[5]Except by pointer aliasing which occurs in many programming languages.

2.3.2.2 Barriers

A *barrier* provides synchronisation between multiple concurrent components. Barriers divide computation into *phases*. At the end of each phase each concurrent component synchronises on the barrier, this synchronisation does not complete until all other components are also synchronising on the barrier. This is similar to a multi-stage bicycle race, with phases representing stages, each stage with a finish line. The cyclists finish a stage at different times; however, all the cyclists start the next stage together. As opposed to mutual exclusion, such as in monitors (2.3.2.1), where components are essentially prevented from executing in the same phase, barriers guarantee that any number of components *are* all executing in the same phase.

A common use of barriers is to synchronise large numbers of components generating results from shared data. An example of this is n-body simulation. Each concurrent component is responsible for calculating the position of a number of bodies. The bodies interact by exerting forces on one another, for example if the bodies are subatomic particles they might attract each other via gravity and repel each other by electromagnetic fields. Each component reads the shared data to discern the influence other bodies have on the bodies it governs then synchronises on a barrier. After completing the barrier the components apply the influences previously calculated before synchronising on the barrier again. This divides computation into two phases, a read phase and a write phase. In the read phase the components can access any shared data in the knowledge that it will not change. In the write phase the components only update shared data they are responsible for in the knowledge that the changes will not collide or interfere. All components advance at the same time in *lock-step*.

The above design pattern can support any number of phases of computation as long as all components are in the same phase; however, to support this, barriers must be carefully designed to separate incoming and outgoing components. It is possible to imagine a scenario where a barrier synchronisation finishes and one component races out of the barrier, finishes its work and returns to synchronise on the barrier before all other components have left. Additionally while one barrier can support multiple

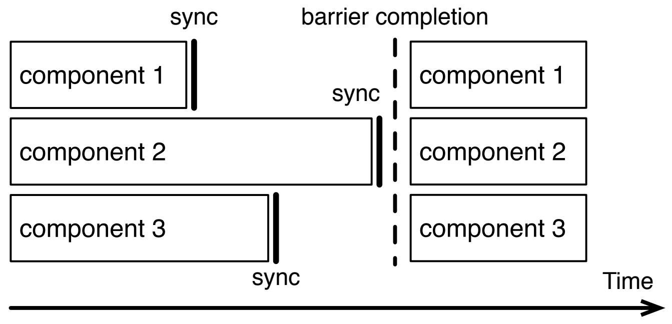


Figure 4: Example of barrier synchronisation. Components arriving at the barrier are blocked. When the last components arrives all components are rescheduled.

phases, it does not allow the identification of the current phase of components, for example if a new component wishes to join the computation part way through. Hence it is often preferable to use a separate barrier for each phase.

Specialisations of barriers such as *clocks* have been proposed to strongly bind barriers with phases [70, 233], much as monitors strongly bind operations in comparison to semaphores. Clocks have distinct phases and can only advance to the next phase when all components are waiting (synchronise) for the clock to advance. It is also possible to observe the current phase of a clock.

It is important to observe that there are two types of barrier:

- static barriers where the number of components synchronising does not change;
- dynamic barriers where components may join or leave the barrier at any time.

Static barriers are relatively simple to manage as a count of waiting or active components. The hazard is that if a component fails, even if failure is detected, the static barrier can never complete. Thus dynamic barriers are desirable for handling failure, but also to allow components to be added during use.

With a dynamic barrier components may *resign* at any point. Care must be taken in the case that resignation would cause the barrier to complete. When a component wishes to use the barrier it must *enroll* on it. Coupled with this the component will need to observe the present phase of the barrier, perhaps using shared data, and potentially synchronise multiple times until the phase is appropriate for the component to begin computation.

Barrier variants exist at all levels of computer software and hardware. At the micro level barrier instructions exist in computer processors to synchronise state with respect to memory in multi-processor systems [143]. At the macro level barriers can be implemented across computer networks and tens of thousands of processors [121]. As with other concurrency primitives, if barriers are used heavily they represent a significant potential overhead. Critically barriers serialise all concurrent components involved at the rate of the slowest. If one component runs significantly slower than the rest, they will spent significant periods of time waiting for the slowest. Essentially the slowest component increases the sequential portion of the computation in Amdahl's law. This limits the suitability of barriers to cases where all components have similar execution times or waiting does not introduce inefficiency. The first case is common when work has been evenly divided among a number of components. The second case applies when there are significantly more concurrent components than physical processors, meaning there is always more work to occupy the time spent waiting for the slowest component.

Partial barriers have been proposed to avoid delays from slow or failed components [27]. With a partial barrier not all components enrolled on the barrier need to reach the barrier in order to complete the barrier synchronisation. This is appropriate when having components in different phases of computation does not endanger the safe operation of the program; a scenario common in distributed systems where data is operated on locally. Components arriving *late* to the barrier will need to be treated specially, adding complexity. Alternatively, if the barriers are part of a choice construct (2.3.3), then a partial barrier need only admit the appropriate number of components to the given choice [260].

2.3.2.3 Interrupts, Signals and Events

A concept originating in computer hardware, an *interrupt* is a signal from outside the computer processor which causes a change in the flow of execution inside the processor. Interrupts exist primarily to simplify the implementation of programs for managing computer hardware such as operating systems. When a piece of hardware needs attention it can send an interrupt, in the form of an electrical signal, to the processor. For example a network interface might interrupt the processor when a new message has been received from the network.

In the absence of interrupts program code must check the state of hardware at regular intervals, a technique called *polling*. Polling is inherently inefficient, as every time a device is polled and does not require attention, the time spent polling is not spent doing meaningful work. Thus if the polling frequency is too high the time spent polling will eclipse the time spent doing useful computation. Conversely if the polling frequency is too low, events which are timing critical may not be dealt with fast enough and errors may occur as a result. This can be likened to putting a kettle on a gas stove and getting on with other tasks in the kitchen. If I do not check the kettle frequently enough it will reach boiling point and eventually boil dry, perhaps damaging the kettle. However if I check it too frequently I will not be able to do anything else in parallel to waiting for the kettle to boil, as essentially I will just be waiting for the kettle, and worse the kettle may never come to the boil [179]. The commonly adopted solution is to place a whistle on the kettle so it notifies or interrupts us when it reaches the boil.

Viewed in a simplified form, the programmer defines a subroutine to be activated on receipt of an interrupt, an *interrupt handler*. The interrupt pre-empts any program code and hence the interrupt handler must save the present state of the processor before performing any work. On completion the interrupt handler clears the interrupt state in the processor to indicate the interrupt has been dealt with, and it may also need to manipulate the device such that it is no longer causing the interrupt before doing this,

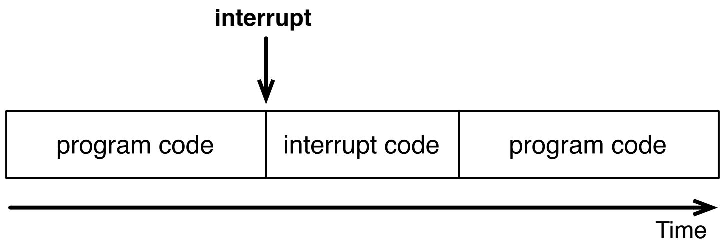


Figure 5: Example of interrupt handling. The execution of the interrupt code is interleaved with the execution of program code.

otherwise the interrupt may trigger again immediately. The interrupt handler then restores the pre-empted program code and execution of continues. The pre-emptive nature of interrupts makes them one of the earliest forms of concurrency as the interrupt is concurrent with program code.

As the execution of the interrupt is arbitrarily interleaved with other program code data-oriented concurrency issues can arise. However, it is not practical for the interrupt handler to synchronise with other concurrent components as they themselves have been pre-empted by the interrupt. To elucidate this, imagine one component takes a lock and begins manipulating data protected by that lock, an interrupt occurs and the interrupt handler wants to manipulate the same data. The interrupt handler can never acquire the lock on the data without restoring the previous component, an operation that is non-trivial or impossible as the interrupt itself has bypassed any scheduling in the running program or computer's operating system. A common solution is to prevent interrupts during so called *critial sections*. This is done by disabling interrupts and denying concurrency. Thus if the programmer wishes to manipulate data which may also be manipulated from an interrupt handler, the interrupt is first disabled and then re-enabled on completion. Even with this mechanism, in a multi-processor system the interrupt may be delivered and handled by another processor.

An alternate solution is to raise the interrupt handler from being a low-level operation to a higher-level concurrent component managed and scheduled with other components in the system. This can be done using the interrupt handler to activate a concurrent component which performs the rest of the work. A split is often necessary as certain actions may need to be taken in the interrupt handler, such as handshaking with hardware while interrupts are disabled, before continuing. Within the Linux kernel these two stages are called top-half and bottom-half. This approach is used in the RMoX operating system to convert hardware interrupts to a message-passing paradigm and remove data-oriented concurrency concerns [49].

The principle of interrupts has been extended to software events within operating systems. An operating system process (2.3.1) can receive a *signal* from the operating system or another process [18, 216, 245]. On arrival the signal interrupts the destination process's current execution and invokes a signal handler which is synonymous with an interrupt handler. This mechanism allows for fault handling, for example when a program attempts to access otherwise inaccessible memory the operating system sends a signal. The default behaviour on receipt of this signal is for the application to terminate; however, a signal handler could be implemented to attempt recovery. While primarily designed for fault handling, as signals can be sent between processes they also provide a form of inter-process communication.

All the same limitations and data-oriented considerations as interrupts exist with this mechanism with the additional consideration that while interrupts are typically synchronised with hardware devices, signals are asynchronous. I have mentioned considerations for asynchronous behaviour with respect to messages in 2.2.2 and will reiterate these in 2.3.2.4. A particular risk with signals is that while messages are generally explicitly received so they do not add concurrency to an existing program, signals pre-empt program execution and create unexpected or undesired concurrency that must be managed. Signals can in some systems even pre-empt signals, meaning the operation of signal handlers themselves can be interleaved. Imagine I am in the kitchen working through a recipe (written on a piece of paper) when suddenly a new recipe (the signal) is placed on top of the current one and I am told to make the new recipe and I cannot return to the old recipe until its done. If yet another signal comes in, I have to start that

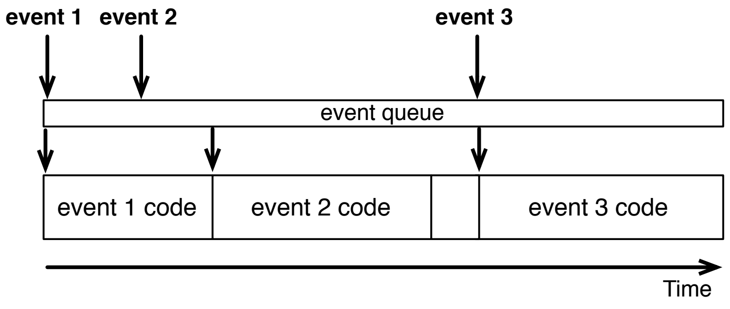


Figure 6: Example of event processing. Incoming events are held in a queue from which they are dispatched.

as well. While I am not preparing these recipes in parallel, they are all concurrently active. Each of these recipes takes more pans and utensils of which there is only a finite number. Also if I happened to be on timing critical step of the first recipe, such as baking cookies in the oven, it might fail if I do not finish the new recipes quick enough, resulting in burnt cookies. This contrived example is an accurate analogy of interrupts and signals.

The concept of an *event* as mentioned in relation to event-driven programming (2.2.3) is synonymous with an interrupt or signal. However, idiomatically events are not pre-emptive. Conceptually the acts of receiving the event and invoking the code to handle the event are separated by a queue. New events are added to the back of the queue, with a scheduler executing the queued events in some order. If the scheduler does not invoke event handlers concurrently then there is no concurrency. While concurrency related hazards are avoided by this method, event handlers must be designed to finish quickly so that new events can be processed in good time. This highlights the advantage concurrency (and pre-emption) can have in maintaining response times while removing concerns from the programmer. The programmer should be able to write a long running computation in one component without having to consider the implications for the response time of other components: this is not possible without concurrency.

In summary, interrupts, signals and to a lesser extent events all introduce concurrency into computer programs. While interrupts may synchronise with hardware this is not a requirement of their functionality, thus they do not inherently provide any synchronisation. Similarly, as the majority of implementations of signals are asynchronous with respect to the sender [18], these do not provide synchronisation either. Interrupts and signals also carry very limited information about their source. Combined with their asynchronous nature, receiving an interrupt or signal is similar to receiving a postcard with the sender's address written on it and the message "call me". The sender does not know when or if they will get a call back and the receiver doesn't know much more than the fact that someone wants their attention. Events, as applied to event-driven programming, can carry data, e.g. "mouse click at coordinates 1,1". This gives events parity with asynchronous messages (2.3.2.4) where the destination is the whole program as a proxy for the event scheduler, which in turn initiates a handler, perhaps as a concurrent component. Critically, interrupts and signals target an entire program (not program component), which must handle them concurrent to any existing operations.

2.3.2.4 Messages, Mailboxes and Channels

A broad concept encompassing many forms of communication is that of *messages*. A message carries data and potentially provides synchronisation. Messages underpin the message-passing paradigm of concurrent programming as described in 2.2.2.

Messages can be categorised by how they are transmitted, received and the medium on which they are carried. Essentially the choice of abstraction chosen for the transmission medium determines the semantics of transmission and reception. In the *actor model* messages are sent via an *ether* with no defined properties [25]. Messages are given an *explicit destination*, which the sender must know before it is able to send the message. Being undefined the ether offers us no guarantees about when or if the message will arrive at its destination, how long it will take or how the message will be ordered with respect to other messages. It is also unspecified as to whether the ether has a storage capacity, i.e. multiple messages can be in moving through the ether at once, or only one

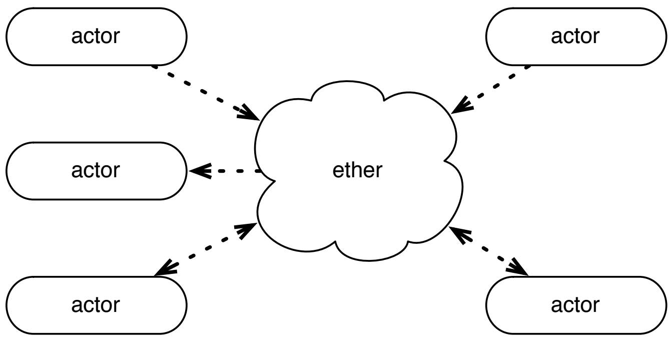


Figure 7: Example of messages in Actor model. Messages travel between actors via the ether.

message can be in the ether at a given time. In the latter case, if the destination is busy when a message arrives the message could be lost, an *asynchronous ether*. Alternatively, the ether might remain full and unusable until the destination is ready, a *synchronous ether*. This would have the knock-on effect of preventing any other messages being sent until the destination is ready.

Mailboxes

As described the actor model ether provides the strongest definition of *asynchronous*, and offers the least guarantees. All other transmission models can be seen as refinements of the ether to add guarantees. Most actor model implementations give each destination or concurrent component a *mailbox* into which new messages are delivered. These mailboxes are typically unbounded in size, preventing message loss if the destination is not ready and the ether is asynchronous, or a full ether if synchronous. It is important to note that a synchronous ether requires less engineering and is hence common where concurrent components are implemented as coroutines or continuations (for example Smalltalk [238]). Thus the need to mitigate the limitations of a synchronous ether and a genuine concern.

While mailboxes may be unbounded in size, all computer systems possess limited memory and storage capacity. Rather than allowing all memory in the system to be consumed by mailbox storage, an upper bound on mailbox size could be defined. If the mailbox is full then it discards newly arriving messages, which could be useful if a runaway component sends out messages endlessly. This could turn a synchronous ether into an asynchronous ether as required. It is an important distinction that the mailbox discards messages not the ether, as this allows the semantics of the discard to be defined. For example the oldest messages in the mailbox could be discarded, meaning the mailbox only contains the most recent messages. Additionally as a mailbox is a store of messages, the component could choose to look through all messages to pick which one to deal with first. This is a form of choice which is covered in more detail in 2.3.3. Essentially mailboxes refine the ether and provide defined behaviour to actor model systems.

Mailboxes separate components from message reception. Components wait for messages by waiting on the mailbox and when not waiting the mailbox prevents incoming messages from interrupting the component (2.3.2.3) or losing messages. With this separation the destination of a message becomes a mailbox rather than a component. Thus it is conceptually feasible that a component could have multiple mailboxes. Within the actor model formalism itself mailboxes are in fact actors themselves delegated with handling messages for another actor [130].

Channels

Thus far messages are created, given an explicit destination, and sent into an ether from which they later emerge at their destination. In principle all components have equal standing within the ether and any component can message any other. Removing the ether removes the communication medium and all components are isolated. In the absence of an ether, explicit communication *channels* must established between components that wish to communicate. With channels the destination is implicit, messages sent in to one *channel end* emerge at the other channel end.

Like mailboxes, channels provide defined behaviour. Where the ether is strongly asynchronous and is refined by mailboxes to provide guarantees, channels can be seen

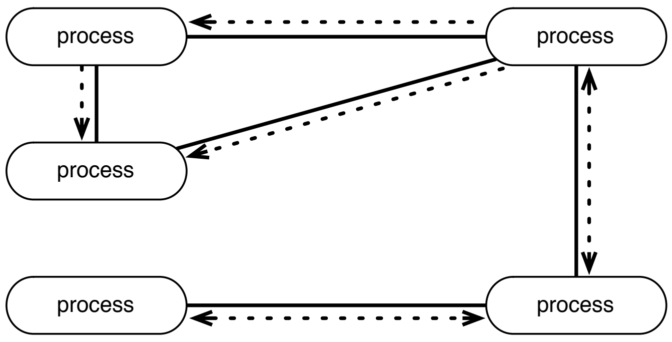


Figure 8: Example of messages with channels. Messages travel between processes over channels.

as strongly synchronous and have their guarantees relaxed to increase functionality. Thus the simplest form of channel is one which possesses no buffering. When a sender attempts to send a message it can only do so if there is a receiver at the other end, if not it must wait. The opposite is also true, since the channel cannot contain a message, the receiver can only receive a message if there is a sender. This provides the strong guarantee that having completed a communication, both the sender and receiver know each other have passed a certain point, and the location of the message. This an implicit barrier (2.3.2.2) between the sender and receiver.

Naïvely synchronisation can be implemented using asynchronous messages by acknowledgements or *handshaking*:

- the sender sends the message, then waits for an acknowledgement from the receiver;
- the receiver on receipt of the messsage sends an acknowledgement to the sender.

However this does not provide the same level of synchronisation as the sender is not aware the message has been delivered until the acknowledgement arrives. As long as the delivery of the acknowledgement is guaranteed this relaxation of synchronisation does not represent an issue, as the observed behaviour of both components having reached a known point is still preserved. Significantly this is based on a consistent ordering of the events as observed by the two communicating parties, an assumption that is not true in an actor model system.

Imagine a system of three concurrent components:

- S, the sender, sends a message $M1$ to R, then waits for an acknowledgement $A1$.
- R, the receiver, receives message $M1$ then sends an acknowledgement $A1$ to S, then sends another message $M2$ to T the third party.
- T, the third party, receives a message $M2$ and sends it onto S.

In this system the expected behaviour is that S sends $M1$ to R, which acknowledges it with $A1$. After sending the acknowledgement, R sends $M2$ to T, which sends it to S. We might expect the S to observe the sequence: $A1$, $M2$; however S can also observe $M2$ followed by $A1$ as messages can be reordered in the ether. In fact this scenario is possible even without the third party if the ether does not respect the ordering of communications between communicating components, something it has no obligation to do, although a guarantee is present in some models (Akka for example 2.5.26.1). The messages sent from R, $A1$ and $M2$, can race in the ether with $M2$ arriving before $A1$.

The solution to the above scenario is to design S so that it does not process any messages until the acknowledgement is received; it must buffer all messages until the acknowledgement arrives. Thus S may have to buffer a potentially unlimited number of messages while waiting for the acknowledgement. This behaviour cannot occur with unbuffered synchronous channels as the act of sending the message and receiving the acknowledgement are indivisible. To generalise, with unbuffered synchronous channels there can only be $\frac{n}{2}$ messages in transit, where n is the number of concurrent components. With asynchronous messaging as per the actor model there is no upper bound on the number of messages in transit or buffered in mailboxes. While with asynchronous messaging it is conceivable that messages could be lost when the system runs out of memory, it is not known how much memory is required for the running system. This contrasts with synchronous messaging over channels where the amount of memory required for the system can be known a priori.

The strong guarantees of channels can be relaxed by making them buffered and hence asynchronous. This buffering could be bounded, such that when the buffer is full the channel degrades to synchronous behaviour or lossy in the same manner previously described for mailboxes. Critically the order of messages within the channel is preserved unless a buffering implementation which reorders messages is chosen. This buffering can be implemented by introducing a concurrent component using two synchronous channels, one to communicate with it, the other to receive from it [228].

As previously discussed, channels have an implicit destination for messages they carry and in the absence of channels an explicit destination embedded in each message must be used. Both of these scenarios must overcome the issue of discoverability, i.e. how does one component know the address of another. Discoverability can be achieved in a number of ways:

1. a priori knowledge written into the program, e.g. the address of a given component is a constant embedded during construction;

2. a directory supported by a priori knowledge of its location, e.g. I can ask the directory for the address of a component (by-name);

3. hierarchical knowledge, e.g. a parent can pass knowledge to its children when they are created;

4. communication, e.g. one component shares its address with others.

Without channels the communication of addresses simply means sending the location of the mailbox of a component. The address of one component is written into a message and sent to another component. With channels the ends of the channel can be communicated independently; this means one component can create a channel and pass both ends of it to other processes. The word *pass* is used to describe communication of channel ends, whereas *copy* is more common for mailbox or actor addresses. This signifies that the end is a entity which if duplicated would become two separate entities, whereas all copies of an address are equal.

When considered as entities, channel ends can be seen to be owned by components. This leads to a distinction between shared and unshared channel ends. An unshared channel end can only be used by the component which owns it. A channel with only unshared ends provides exclusive communication between components. Conceptually the sharing of channel ends is simpler if channels are unidirectional with sending and receiving ends. Sharing the sending end allows an interleaving of messages from senders to receiver, as many people can send letters to a single company. Sharing the receiving end scatters delivery of message across a set of receivers, as a company may employ many people to open and sort letters. An actor model system with mailboxes can be modelled as system in which each component has a channel, where the sending end is shared with all other components. A component messages another using the appropriate sending end and receives messages using its unshared receiving end.

Summary

In summary messages carry data between components and can provide synchronisation. Explicitly addressed asynchronous messages based on actor model semantics are present in many programming languages, notably Erlang (2.5.11), Scala (2.5.26), and Clojure (2.5.9). Smalltalk, a significant influence on the actor model itself, provides addressed synchronous messaging (2.5.27). Channels with both synchronous and asynchronous behaviour are present in other languages, notably occam (2.5.21), Google Go (2.5.13), and XMOS XC (2.5.30). Unbuffered primitives such as channels carrying messages can be used for synchronisation, although handshaking (with weaker guarantees) can simulate this with asynchronous messages. Critically there is duality between programs constructed using monitors (or semaphores) and programs constructed using messages [159]. However it is important to select the approach which most accurately expresses the model (2.1.1).

2.3.2.5 Promises and Futures

A *future* is a value which is not yet ready (finished computing), but will be at some point in the future [44]. A *promise* is the same as a future, although the name may suggest more strongly that the value *will* be computed [227]. This is important because futures are heavily associated with *lazy evaluation*.

Lazy evaluation describes the concept that given an expression such as $y = x + 42$, I only need compute y if it is actually used. This applies through dependency, so if I define $z = y - 1$, but never use z, then I need not compute y or z. In a pure functional language, such as Haskell (2.5.15), where computation does not have side-effects then all computation can be lazy as the order in which steps are taken does not effect the result. This is like executing a recipe backwards and whenever a item is mentioned that is missing, I read further back to find how to make it. For example (sponge cake):

1. After 25 minutes take cake out of oven. *I need a cake.*
2. Place cake mixture into tin, then place cake into oven for 25 minutes. *I need a tin and cake mixture.*
3. Combine flour with egg mixture to make cake mixture. *I need flour and egg mixture.*
4. Beat egg into creamed butter to make egg mixture. *I need egg and creamed butter.*
5. Cream butter and sugar to make creamed butter. *I need butter and sugar.*

At the point a promised value is needed the future must be *resolved*. This means either the value must be computed (*lazy future*), or it must be waited for. This action can be explicitly requested or implicit in use of the future. In lazy evaluation the creation and resolution of futures, called *thunks*, is implicit and is a kind of closure (2.3.1.3). A wait may occur if the future is being computed concurrently. Concurrency is possible because the computation of a future can begin at the moment it is created (a *concurrent future*). If computation completes before the value is required then no wait is induced, otherwise the resolving component synchronises with the computation. Assuming the computation has no side-effects the result of both these scenarios is the same.

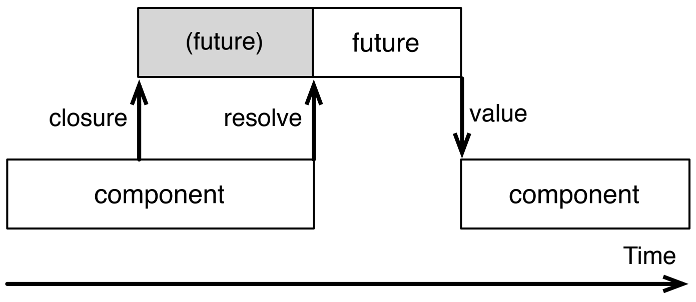


Figure 9: Example of futures and resolution. A component creates a closure which becomes the future, at this point the future can be active concurrent to the component. Later the component resolves the future which blocks the component until the future is computed and produces a value.

Promises can be pipelined to improve performance [163]. If operations on futures are themselves futures the new future can be passed to further steps without delaying computation. This can be imagined to be a sort of reverse pass the parcel, at each step the parcel is wrapped again together with instructions of what to do with its contents (like a closure). Conceptually the parcel then unwraps itself from the inside as its components become ready. The parcel can continue to be passed around and more operations added. Eventually this magic parcel will turn into the finished result wherever in the system it ends up.

For performance reasons it may also be useful to test a future to see if it is ready without resolving its value. This technique is often called a *read-only view*, because it does not change or write the state of the future. If a read-only view is not available then any attempt to observe the value of a future will force resolution of its value. This relates to choice (2.3.3), as it allows alternate actions to be taken if a future is not ready.

Resolution may itself be synchronous or asynchronous. Synchronous resolution is as previously described, the observer has to wait for the value or perform the computation. In asynchronous resolution a message (or other signal) it sent to the future requesting the value, and when the value is ready it is sent back. During the intervening period the observer can perform other tasks.

A concurrent future can be seen as a special case of a channel which only ever carries one result. The computation component holds the sending end of the channel and sends the result when computed. The observer holds the receiving end from which is reads from when the future needs to be resolved. If the result is ready then it will be received, if not the observer will wait until it is. A read-only view can be implemented if the channel can be tested without waiting.

2.3.3 Choice

The *if* statement or similar is present in every Turing complete programming language. This tests a *boolean* (true or false) expression, and takes one action if it is true or another if it is false. For example, if the cookies are golden brown, then take them out of the oven, else wait a few more minutes. An if statement *branches* the *state* of the program. Notionally there are three states: *P* the state prior to testing, *T* the state if the condition is true and *F* the state if the condition is false. This can be seen in figure 10.

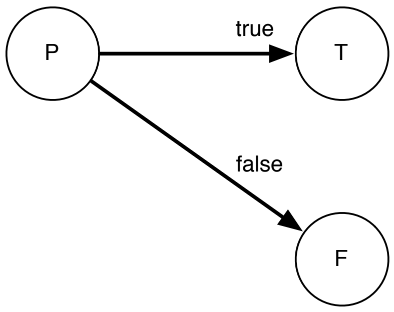


Figure 10: An if statement branches state into true and false states.

By testing multiple conditions in turn a selection can be made between any number of states. This is a low-level mechanical view of the high-level concept of *choice*. Figure 11 compares multiple if branches to a single choice of a number of possibilities. While the number of resulting states is the same, the multiple test and branch operations lack atomicity, which is to say the operation is observably divisible into multiple operations. As there are multiple operations a sequence between these tests is created. In the absence of concurrency this sequence does not affect the result; however, in a concurrent system the state of other components can change between sequence steps. Imagine I ask a question of three people, with fairly boring names like *A*, *B* and *C*. I want to take the answer of the first who is ready to answer. First I look at *A* and they are not ready, so next I look at *B* who is not ready, then I look at *C* who is ready, so I take *C*'s answer. It is possible for *A* to become ready then *C* to become ready while I am looking at *B*, so while I pick *C* in fact *A* was first. Whether this difference in behaviour is significant to correctness varies, however the critical observation is the desired behaviour must be captured by a high-level concept such as choice rather than simple branching.

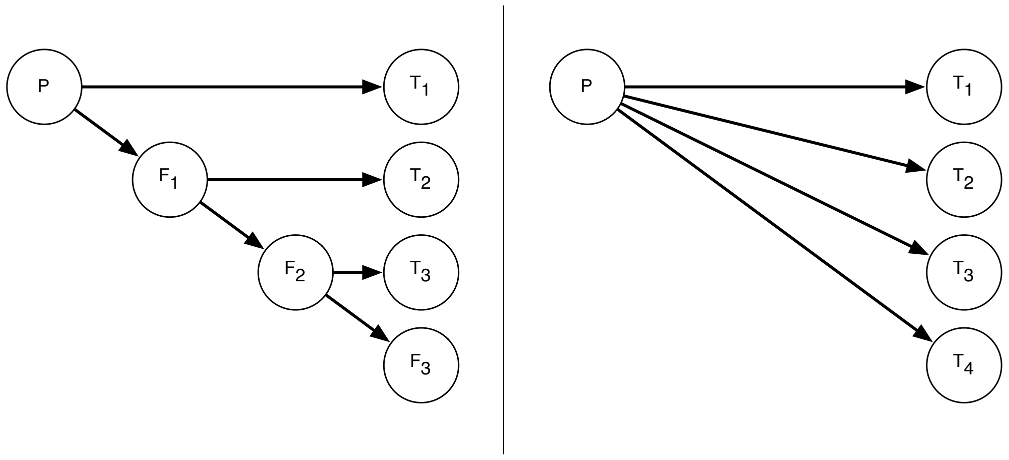


Figure 11: Multiple if branches are equivalent to a single choice with respect to end states, but if statements lack atomicity.

2.3.3.1 Determinism

A program is *deterministic* if it is possible to determine the output of the program solely from its input (and knowledge of its operation). The output of the program will always be the same for the same input data. A program which does not admit concurrency is always deterministic. Even without being internally concurrent a program can admit concurrency if its behaviour can be changed by external events, for example if it accepts user input at arbitrary points or makes use of time by reading the system clock. The output of the program is now not only determined by *what* is input, but also *when*. Such a program typically is *non-deterministic*. Without a precise model of the behaviour of all external events, the state and output of the program cannot be determined. Such a model is in general unavailable, particularly when order and time are involved. This is not least due to time being relativistic and generally infinitely divisible [101].

As noted, concurrency can introduce non-determinism into computer programs [226]. The order of computation can change dependent on the order and timing of external events, and the order that concurrent components are scheduled and executed by computational elements. The goal of *expression* (2.1.1) is to capture and control non-determinism so that the state of the program can be reasoned about. This contrasts with *performance* (2.1.2) which seeks to admit non-determinism in order to increase the possible paths to a solution. Ultimately a balance is required so that the program produces a valid result as efficiently as possible.

To take a more concrete example, recall how we were making marinaded steak (2.1.3): "we cannot fry the beef before it has been marinaded, and we cannot marinade the beef before we have made the marinade; however, you can make the marinade while I go and buy the beef". There are four events:

A make marinade

B buy beef

C marinade beef

D fry beef

D must follow C and C must follow A and B, but the order of A and B does not matter. Thus by not determining the order of A and B we allow two different orders to be valid solutions: A, B, C, D or B, A, C, D. One of these may be more efficient and hence we can use that order to maximise performance. Importantly A and B both take time so we are also allowing them to arbitrarily overlap.

Choice over concurrent events both admits and controls non-determinism. If we wait for an event A then another event B, and we only allow this order, behaviour is deterministic (assuming A and B are themselves deterministic). By allowing a choice over A and B these events can be reordered, admitting non-determinism. If we give *priority* to A over B and both A and B occur or are ready simultaneously then A will always be chosen, controlling non-determinism.

2.3.3.2 Programming Language Support

A variety of choice functions are available in concurrent programming languages allowing a program to select from a set of communication or synchronisation events. These primitives provide *choice over events*. Choice over events implies waiting for events to be invoked by other components. This also implies a *race* between those components as to which will invoked first and hence which event will be ready first. This race is the source of non-determinism which is admitted into the component making the choice.

It is important to observe that while it is theoretically possible to have choice over semaphores and monitors it is not implemented in practice. This is in part due to semaphores and monitors principle use as mutual exclusion mechanisms. They are invoked because access to a protected resource is required for the program to continue, and do not directly model events. Mutual exclusion serialises operations and as such the race occurs in the order of components reaching the exclusive region. Within the region a component can make decisions based on the protected state which will not change for the duration of the region. This is *choice over state*, the same as if statements, with the non-determinism occurring outside the mutually exclusive region. This moves control of non-determinism from the choice to the mutual exclusion primitive. If a priority is to be applied to operations (choice over events), it is the semaphore or monitor that must control this. To achieve this the semaphore or monitor needs to be made aware of the operations. A choice over events in the semaphore or monitor will need to work in tandem with a choice over state in concurrent components. From the standpoint of clarity of expression this can be an undesirable duplication.

Interrupts and signals can be seen as a continuous choice between the running computation and the interrupt. Priorities can be specified such that a high priority interrupt can occur during a low priority interrupt, but a low priority interrupt cannot occur during a high priority interrupt [18]. Interrupts are a priori declaration of choice. Without explicit engineering it is not possible for a component to know or use knowledge of such a choice occurring.

Within Actor model style message passing, if a mailbox mechanism is used then the mailbox can provide choice (over event reception). Message events arrive in the mailbox where they are buffered. The component can search through the mailbox and select particular types of message to process, or if there are no messages of the desired type then wait for them to arrive. Since all events arrive via the single mailbox there is no choice over primitives, rather the primitive makes the choice on the components' behalf. As the mailbox is asynchronous these events do not entail any *commitment*; more specifically the messages have already been received so accessing them from the mailbox does not commit the component to synchronisation with other components unless the component is explicitly engineered to do so. This pattern is used in many actor model style implementations. Erlang (2.5.11) and Scala (2.5.26) have mailbox pattern matching. Akka (2.5.26.1) allows arbitary definition of message receipt handling, including mailboxes. However implementations such as Clojure (2.5.9) do not provide this functionality although it can be implemented.

Operating systems demonstrate an alternative to mailboxes for choice over asynchronous primitives. The POSIX operations *select* and *poll* allow a program to test a number of *file descriptors* to see if they are ready for input or output (or whether an error has occurred) [18]. A file descriptor can represent an open file, a system device, a connection to another program on the same system, or a connection over a network. These operations wait for a primitive to become ready or a timeout to occur. File descriptors are often considered synchronous; however, this is with respect to the underlying buffers. A file descriptor which is ready for input represents that the underlying buffer contains data that is ready. Ready for output means there is space in the underlying buffer to hold output. This makes file descriptors equivalent to buffered channels and essentially asynchronous.

2.3.3.3 Commitment

With synchronisation primitives such as synchronous messages (with or without channels) or barriers, choice carries with it the notion of *commitment*. If a given option is selected the component synchronises with other components involved in that primitive. This is significant as the decision of one component implicitly affects other components. Consider two groups of people, X and Y, waiting for an individual A. If A chooses to go with one group, say X, then the other group (Y) will still be waiting. This behaviour is easy to understand; however, the result becomes more complex if more than one person (component) is making a choice. Imagine another person, B, is also potentially involved in both groups and neither group will be complete unless both A and B choose to go with it. Without communication between A and B, they could select opposing groups and both groups will wait forever: this is a deadlock.

This scenario can be solved in one of two ways: consult a shared arbiter who makes the selection in such a manner as to avoid deadlock, or separate choice into multiple phases of commitment so as to implement a consensus algorithm [256]. Broadly a consensus algorithm involves all components marking the options they are waiting on, and in doing so if one option gains enough participants then those waiting on that option are woken up. All components must then remove marks from other options, and if in doing so a component discovers another option has become ready then that component must remove all its marks and wake up any component waiting on those options before restarting from the beginning. Having removed all other marks a component waits on the committed option for all other commiting components to synchronise. Because this algorithm can restart any number of times, it can potentially take an infinite amount of time to complete. A shared arbiter does not have this disadvantage; however, it represents a bottleneck and single point of failure in the system.

In addition to the need for consensus between choices there is the possibility of conflicting priorities, e.g. A prioritises group X while B prioritises group Y. It is not possible to proceed without violating one of the party's priorities. This could be considered a deadlock as neither components request can be *correctly* satisfied, and hence a design failure[6].

Languages allowing choice between synchronous primitives must address both consensus and priority concerns. The occam language does not allow barriers or output in its *alternation* (choice) primitives (2.5.21). This means that output and barrier synchronisation are always fully committed operations. A component selecting over inputs can make a completely local selection knowing that the sender has committed and will not backout of their committment. In Google Go the *select* statement does not allow priority, but selects arbitarily when multiple options are ready. Simplistically consensus is implemented by locking channels before making a selection such that the state cannot change during the choice process (providing atomicity), however it is a costly operation. The JCSP library for Java provides choice over all primitives using a global arbiter mechanism called an *oracle* [258, 256]. A generalised choice mechanism without an oracle has also been developed for Communicating Scala Objects [165].

As previously mentioned (2.3.2.5), futures can be considered channels which only produce one message. Thus futures can be incorporated into choice in the same manner as channels. The future will always commit to output, simplifying its integration. Specifically a future only has three states: *not ready*, *ready* and *read*. Transitions between states are monotonic so once a future is in a *ready* or *read* state it will never return to the *not ready* state.

[6]This scenario can be mitigated if tests for availability are used, as conflicting scenario will not be committed [164].

2.3.3.4 Time

In addition to the primitives mentioned it is desirable to allow a choice over timing events. For example making a choice between a mailbox and a relative point in time provides a timeout, i.e. if there are no appropriate messages in the next period then take an alternate action. Choice between points in time can be resolved a priori: the earliest point in time will become ready first. Choice between a synchronisation primitive and time is simpler than generalised choice between synchronisation primitives. Similar to state transitions with futures, time is monotonic: a given point has either passed or not.

2.3.3.5 Summary

Choice is important for allowing non-deterministic behaviour. Data-oriented concurrency uses implicit *choice over state*. This is often indistinguishable from choice in a program without concurrency. Message-passing concurrency uses explicit *choice over events*. When choice is made between synchronising events such that circular dependencies could occur, then *consensus* must be reached through internal or external mechanisms.

2.4 Process-oriented Programming

The term process-oriented programming originates in 1992 [102, 263] as a means to describe the style of programming associated with CSP-inspired languages such as occam. Definitions of process-oriented programming have varied; however, this thesis uses Sampson's extensive definition [228]. Sampson defines five important aspects of process-oriented programming: *concurrency*, *isolation*, *communication*, *composition* and *reasoning*.

Concurrency, its implications and uses have been described in previous sections. Process-oriented programming provides a single primitive for concurrent program components, the *process*. Processes execute in *isolation*, meaning they do not share resources.

Where sharing is required it is accomplished by explicit *communication*; this follows the mantra "do not communicate by sharing, share by communicating". Isolation and communication make interactions between processes explicit and prevent many problems associated with data-oriented concurrency (2.2.1). As previously noted communication encompasses synchronisation. A process may use any number of communication and synchronisation primitives; this distinguishes process-oriented programming from the Actor model (2.2.2) where an actor can only operate on messages explicitly sent to it.

In the process-oriented programming style software is constructed through the parallel *composition* of processes. These form a *process network*, a graph where processes are the nodes and communication links between them are the edges. Each process may be internally concurrent with so called *subprocesses*, with such details hidden from the overall system through isolation. This allows a hierarchical model of composition which bears similarity to naturally occurring structures. For example the human body is composed of organs, each of which has inner components composed of cells, composed of atoms, composed of subatomic particles. Just as it is possible to reason about the behaviour of an organ in terms of its external behaviour and function, so it is possible to reason about processes based on their external behaviour with respect to communication and synchronisation primitives irrespective of internal processes. For process-oriented programming this *reasoning* is aided by process-calculi (2.2.2), in particular CSP [134, 226] and the pi-calculus [184]. The compatibility between process-oriented programming and process-calculi is in part due to the propagation of inspiration and concept between the two [172, 225, 226]. This allows manual or partially automated proving of properties of process-oriented programs and design patterns, such as deadlock freedom [259].

Process-oriented programming advocates the use of concurrency for program structure: a philosophy with widespread support [37, 202, 228]. It is distinguished by the use of synchronous rather than asynchronous communication and the use of implicit addressing via channels. Synchronous communication aids in both formal and informal reasoning. Implicit addressing aids abstraction as the programmer uses the interface (or protocol) of a channel separate from how its operations are implemented by a process or processes [228]. This is an equivalent concept to classes in object-oriented programming [86].

2.4.1 Primitives

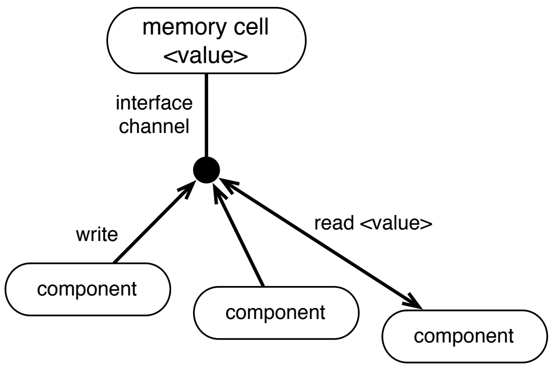
At its simplest, process-oriented programming consists of only concurrent *processes* (2.3.1) and synchronous unbuffered unidirectional *channels* (2.3.2.4) with *choice* (2.3.3). Each channel has two ends, one for readers who receive messages from the channel, one for writers who send messages to the channel. Static process-network channels need only be point-to-point, that is to say there is only one possible reader or writer of a channel at a given time. For dynamic process networks shared channels are required, i.e. a channel can have multiple readers and writers simultaneously. If a channel is shared, messages must be indivisible such that two messages sent simultaneously, A and B, are received as distinctly A then B or B then A, not a combination of A and B. Messages may be of any size, but the minimum size must support the transmission of a channel end. While a channel may have multiple readers each message sent along the channel is received by exactly one of those readers; which one is an arbitrary decision. Additionally when multiple writers attempt to use a channel at the same time which one proceeds first is also an arbitrary decision. If the progress of all components in the system is to be guaranteed the aforementioned arbitrary decisions should guarantee that no reader or writer will wait for ever. This is most simply achieved by the use of first-in first-out (FIFO) queues for readers and writers.

Given this definition it is (informally) possible to construct all other concurrency primitives.

2.4.1.1 Asynchronous Channels

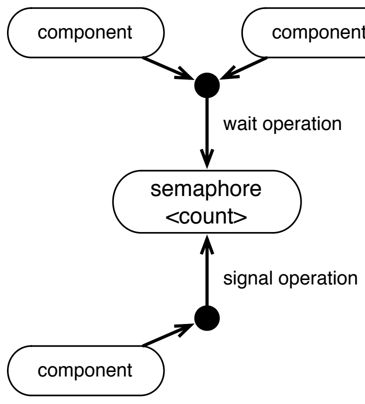
Asynchronous channels can be constructed from two synchronous channels and a process, the *channel process*. Writers use one channel A, the reading end of which is connected to the channel process. Readers use the other channel B, the writing end of which is connected to the channel process. The channel process moves messages between A and B dependent on the desired behaviour. This behaviour will typically be some form of buffer, creating a *buffered channel* [228].

2.4.1.2 Shared Memory



Each unit of shared memory is represented by a process which holds the shared data, a *memory cell*. The memory cell receives request messages on an interface channel. The writing end of the interface channel is shared amongst the users of the shared memory. Requests to read or write the shared memory are sent as messages down the interface channel. A read request message contains a response channel end on which the result is sent. The memory cell enacts requests on the shared data. This model can be extended to support atomic operations as the interface channel serialises all requests. The memory can be made asynchronous if a buffered channel is used for the interface channel. Notationally this is the same model used to implement shared memory on modern computer systems (2.6).

2.4.1.3 Semaphores



A semaphore can be constructed from one process and two synchronous channels. The *semaphore process* maintains the semaphore count. One channel carries *P* (wait) operations, and the other channel carries *V* (signal) operations. A message on the *P* channel reduces the count, whilst a message on the *V* channel increases the count. The contents of the messages is irrelevant. While the count is greater than zero the semaphore process makes a choice between the two operation channels. If the count is zero or negative then only operations on the *V* channel are received; this means *P* operations will wait.

2.4.1.4 Monitors

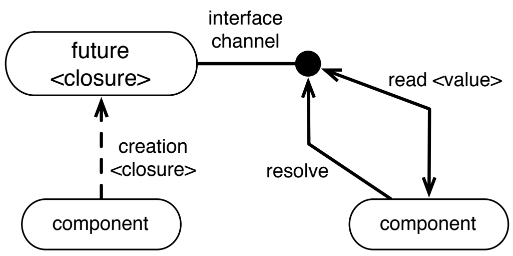
Essentially a process performs many of the functions of a high-level monitor by design. A process provides mutual exclusion of operations on its state (ignoring subprocesses), as it explicitly controls when operations are invoked through the reception of messages. The monitor primitive assumes that control flow moves into and out of the monitor. For concurrent processes this assumption is not true, hence removing the need for much of a monitor's functionality. Looking at Hoare's work, monitors can in a sense be seen as an early version of processes [133, 134]. If monitor-style conditions are required, these can be implemented within the process receiving operations, the *monitor process*. Each request to the monitor process being contains a response channel end. If the request cannot complete immediately the monitor process queues it internally until its conditions are met. On completion of the request the monitor process signals the corresponding response channel end by writing a message to it.

2.4.1.5 Actor Model

An Actor model style system can be modelled by considering each actor as a process, an *actor process*. Each actor process has a reading channel end on which it receives messages from the ether (*ether receive channel*), and a writing channel end on which it sends messages to the ether (*ether send channel*). The ether send channel is shared by all actor processes, the ether receive channel is unique to a given actor.

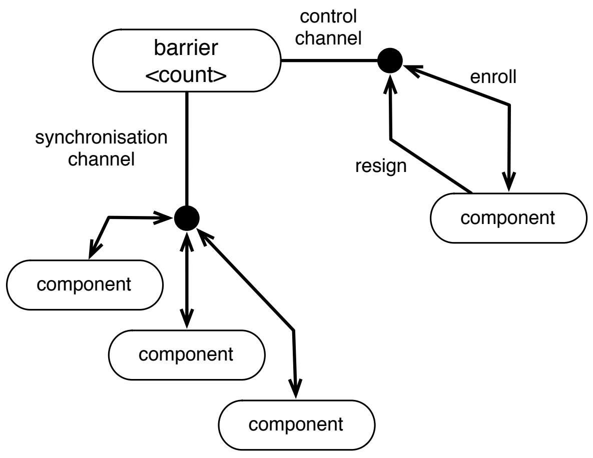
An *ether process* holds the writing ends of all ether receive channels, and the reading end of the ether send channel. It reads messages from the ether send channel and routes them to the appropriate ether receive channel based on their embedded addresses. The ether receive channels can provide common behaviours, for example they could be buffered channels creating a buffered ether. Alternatively the ether receive channels could be supplemented with processes which implement mailbox functionality such as searching over available messages. The ether process could also provide routing between distributed computer systems each of which has its own ether process in a model similar to Erlang (2.5.11).

2.4.1.6 Futures



A future can be implemented by a process and a channel. At the point the future is created its closure (2.3.1.3) is passed to a new process. This *future process* computes the future result and sends it down the channel. To resolve the result of the future the creator simply reads the future channel and on receiving the result replaces the channel with the result. Alternatively the future process might not begin computation of the future until requested to do so. The future channel direction is reversed, so that when the creator wishes to resolve the future it requests the result by sending a message with a response channel on the future channel. The future process computes the result and sends it on the response channel. This design enables lazy evaluation of futures. The future process could store the result and provide the stored result on future requests, allowing shared futures. In principle either of these designs allow future pipelining, as the future channel can be passed to the closure of another future.

2.4.1.7 Barriers



A barrier is similar to a semaphore process. If the count of processes synchronising on the barrier is a constant N, then a barrier can be implemented with one process and two channels, *signal* and *wait*. The *barrier process* receives N messages from the *signal* channel, before sending N messages on the *wait* channel. Each process synchronising on the barrier sends one message on the *signal* channel, before waiting for one message on the *wait* channel. Thus the barrier process will not release any waiting processes until it has received all signals. This requires synchronous channels to prevent sends on the *signal* channel completing while the barrier process is still sending messages on the *wait* channel.

Alternatively, for a dynamic barrier the *barrier process* maintains two counts: the number of processes enrolled on the barrier, the number of processes synchronising on the barrier. It has a single operation channel, on which is receives three types of request:

- an *enroll* request increases the enrolled process count,
- a *resign* request decreases the enrolled process count,
- a *synchronise* request increases the count of processes synchronising on the barrier.

With each synchronise request is passed a response channel. If the counts are non-zero and become equal through a resign request or a synchronise request, then the synchronising count is reset to zero and the barrier process sends a message on all response channels before discarding them.

2.5 Concurrency Support in Programming Languages

This section surveys concurrent programming styles and primitives available in programming languages and support libraries. Languages in this section were selected on the basis of popularity (top five languages in TIOBE index [246]), or if they have built-in support for concurrent programming or a prescribed concurrent programming style. Each language section begins with table of key facts which are comparable between languages.

- **Type** being one of either *language*, *library* or *interface*. A *language* is a complete programming language. A *library* is a set of predefined operations and runtime support which integrates with one or more languages, but is not built into a specific language. An *interface* is a set of standardised operations (API) which may be incorporated into a language or provided to a language through a library, e.g. MPI or POSIX.
- **Year** of first announcement or release. Although development will technically have begun in years prior.

- **Origin** of the language or library, categorised as the type of institution primarily responsible for development of the language. This is a combination of *industry* used to defined any commercial enterprise, *government* funded research and development (e.g. defence sector), *academia* (i.e. university funded), or *community* project (typically open-source).
- **Popularity** ranking as defined by TIOBE index 2013, see 2.7.1.
- **Keywords** which describe features of the language, see 2.5.1.

2.5.1 Keywords

The keywords used to describe languages in this section are defined as follows.

- **actor model**: directly cites inspiration from the Actor model, see 2.2.2.
- **channels**: has channels, see 2.3.2.4.
- **C-syntax**: use a syntax derived from the *C* programming language.
- **CSP**: directly cites inspiration from CSP, see 2.2.2.
- **data-oriented**: focuses on data-oriented concurrency, see 2.2.1.
- **functional**: is intended for functional programming, see 2.1 and 2.2.3.
- **futures**: has futures, see 2.3.2.5.
- **lazy**: uses or supports lazy evaluation, see 2.3.2.5.
- **mailboxes**: has mailboxes with asynchronous messaging, see 2.3.2.4.
- **messages**: has generalised messages which are not directly related to Actor model, channels or mailboxes.
- **monitors**: provides monitors, see 2.3.2.1.
- **OO**: is intended for use with object-oriented structures, see 2.1.

- **PGAS**: uses a partitioned global address space model, see 2.2.1.
- **semaphores**: provides semaphores, see 2.3.2.1.
- **vectorising**: uses vector techniques to implement parallel processing, see 2.2.1.

2.5.2 Ada

Type:	language
Origin:	government
Year:	1980
Popularity:	20
Keywords:	*CSP*, *OO*, *POP*

Ada was developed as a unifying language by the US Department of Defense to reduce the number of languages in use in defense projects [137]. Due to its properties it has found use in real-time and safety critical applications such as air traffic control systems [105].

Ada has a concurrency model with a concurrent components called *tasks*, which are essentially coroutines (2.3.1.3) [82]. Tasks can synchronise with other tasks through defined entry points (inspired by CSP events [226]). These entry points can carry data, which makes them a form of explicitly addressed synchronous message passing (2.2.2, 2.3.2.4). Choice over entry points is possible and can be combined with time delays. Entry points provide *extended rendezvous*, a form of synchronisation where control is passed through the entry point and does not return until the task that has been invoked is ready. This means that while Ada does not have channels, tasks can be used to implement synchronous and buffered channels. Thus Ada supports process-oriented programming in principle.

2.5.3 Alice

Type:	language
Origin:	academia
Year:	2000
Keywords:	*data-oriented*, *functional*, *futures*, *lazy*

The Alice programming language is an extension of Standard ML designed to support logic (fifth generation) programming [155, 231]. This is achieved through introduction of promises and explicitly lazy evaluation [227]. Expressions explicitly designated lazy create a future, and a promise is an explicit future. Additionally an arbitrary closure can be declared concurrent. Alice adds a data-oriented (graph reduction) model of functional concurrency to Standard ML, whereas Concurrent ML (2.5.10) adds a message-passing model.

2.5.4 C

Type:	language
Origin:	industry (Bell Labs)
Year:	1973
Popularity:	1
Keywords:	*C-syntax*

The C programming language is included here only to give complete coverage of the top five most popular programming languages. C does not have any embedded concurrency support, although it is commonly used with POSIX threads (2.5.24). It was intended as a high-level language for implementing operating systems which were at the time still developed in assembly language [208]. For this reason it was designed so as to have an efficient mapping into machine instructions [215]. This has often led C to be referred to as a high-level assembly language.

2.5.5 C#

Type:	language
Origin:	industry (Microsoft)
Year:	2000
Popularity:	5
Keywords:	*C-syntax, data-oriented, functional, futures, monitors, OO, semaphores*

C# can be seen as Microsoft's response to the release and relative success of Java. Both languages take a C-syntax and add object-oriented programming features, automatic memory management and a managed runtime environment (see 2.5.17 for details). Over time C# has developed to include functional programming features and support *task-based* asynchronous programming. The asynchronous programming model is essentially futures with a programming style similar to that of Cilk (2.5.8). C# also includes common data-oriented concurrency primitives such as semaphores and monitors.

An extension called Polyphonic C# adds concurrency based on the join-calculus [55]. Critically this adds asynchronous method calls and so called *active objects* [160]. Synchronisation patterns called *chords* can be used to connect asynchronous methods. A chord cannot complete until its component synchronisations are performed. Work on Polyphonic C# has moved to a new language Cω (C Omega) [56].

2.5.6 C++

Type:	language
Origin:	academia / industry
Year:	1983
Popularity:	4
Keywords:	*C-syntax, data-oriented, futures, OO, semaphores*

The C++ programming language was designed as an object-oriented extension to the C programming language. The C language was used for its performance and low-level access, while objected-oriented features were incorporated from SIMULA (an ALGOL derivative). In addition to an object-oriented programming style, one of C++'s defining features is its standard template library. This provides generic algorithms that save programmers from rewriting or incorporating their own implementations of commonly used programming patterns.

C++ has continued to develop and new standards are defined approximately every four to five years. While the original language contained no concurrency model, the current direction in software development has lead the most recent revisions of the language and standard library specifications to address concurrent programming [20, 17]. Amongst these changes are the introduction of a memory model which specifies the behaviour programmers should expect when using shared memory with potential races; this mirrors work done on the Java memory model (2.5.17). The C++ standard library now contains atomic variables, threads, semaphores, futures and condition variables. These features in general mirror those supported by the POSIX threads interface (2.5.24). An extended standard library called Boost adds message queues which are similar to channels as well as barriers [22]. However choice over these and other primitives is not supported.

2.5.7 Chapel

Type:	language
Origin:	industry (Cray)
Year:	2004
Keywords:	*data-oriented*, *futures*, *OO*, *PGAS*

Chapel is an explicitly parallel programming language developed by Cray as part of their participation in DARPA's High Productivity Computing Systems program [68, 4]. Chapel has no notion of threads of execution, but has subcomputations or *tasks* which may be executed concurrently [64, 67]. In line with the partitioned global address space model, Chapel allows the placement of tasks and their data in a *locale* (part of the computer system); however, data can be accessed between locales. Tasks can synchronise through a *sync* statement, a form of barrier (2.3.2.2), but cannot choose between synchronisations. Data can be made atomic such that operations upon it are serialised. Critically Chapel has sync variables, these can be single use where reads wait until a write occurs which may only happen once, in which case these provide *futures*. Alternatively in general use reads empty the variable allowing further writes. Writes wait if the variable is already full; in this use sync variables provide single place buffered channels. Without choice (2.3.3), Chapel admits only implicit non-determinism through the order in which operations access sync variables.

2.5.8 Cilk

Type:	language
Origin:	academia
Year:	1994
Keywords:	*C-syntax*, *futures*

Cilk adds to the C language a concurrency model based on tasks with well-structured computational dependencies (directed acyclic graphs) [58]. The programmer *spawns* concurrent tasks which are functions that return a result, and a *sync* operation can be used to wait for these functions to complete. Tasks are essentially futures with the sync operation forcing resolution. Additionally *inlets* add custom synchronisation between a task and other tasks it has spawned.

The novelty of Cilk is its use of a last-in first-out (LIFO) scheduler to execute its simple *fork/join* model. Such a scheduler can be efficiently implemented on a multiprocessor computer using work-stealing (see Chapter 3). This technique has been replicated in other languages such as Java and X10 [161, 79]. Intel provides a commercial implementation of Cilk alongside their Thread Building Blocks library (2.5.16).

2.5.9 Clojure

Type:	language
Origin:	academia
Year:	2007
Keywords:	*data-oriented*, *functional*

Clojure is a Lisp (2.1) inspired functional programming language which runs on top of Java's virtual machine (see 2.5.17) [131]. This allows Clojure to integrate with existing Java code and access Java's standard library, an approach similar to that taken by Scala (2.5.26). A pure functional model with immutable state removes many data-oriented concurrency concerns from the language. Software transactional memory (2.2.1.2) is used where state needs to be shared between concurrent tasks. Clojure also supports concurrent *agents*; however, these are reactive and not autonomous, and as such can be seen as continuations. Java's concurrency primitives such as message queues can be used to implement communication channels between agents.

2.5.10 Concurrent ML

Type:	library
Origin:	academia
Year:	1996
Keywords:	*channels*, *functional*, *messages*

Concurrent ML is a extension to Standard ML which adds lightweight concurrency primitives [212]. Synchronous and asynchronous channels are supported, along with choice over channels and timers. This means Concurrent ML supports a process-oriented style of programming. Multiprocessor execution is not supported so Concurrent ML's concurrency is for *expression* not *performance*, although a parallel implementation has recently been developed [211]. Work on concurrency in ML has largely moved on to Manticore (2.5.18).

2.5.11 Erlang

Type:	language
Origin:	industry (Ericsson)
Year:	1986
Popularity:	34
Keywords:	*actor model*, *functional*, *mailboxes*

The Erlang programming language originated in the telecommunication sector where it was designed for implementing concurrent programs that ran, potentially, forever [38]. Continual operation is supported by the ability to replace code on the fly. Concurrent processes with isolated state and communication by asynchronous message passing ease the construction of programs to support code replacement at runtime.

Process mailboxes are the endpoints for all communication. Any process can establish communication with another if it holds its identifier. Identifiers themselves can be communicated allowing arbitary communication graphs to be constructed. Each process has a single mailbox which it can poll or wait on for messages. When a process attempts message receipt it can specify matching patterns (filters or priorities) based on message elements.

2.5.12 Fortress

Type:	language
Origin:	industry (Sun)
Year:	2004
Keywords:	*data-oriented*, *futures*

Fortress, like Chapel (2.5.7) and X10 (2.5.29), was developed as part of Sun's involvement in DARPA's High Productivity Computing Systems (HPCS) program [4]. However development has now ceased due to Sun being dropped from the HPCS program and Sun's sale to Oracle. Fortress was intended as a "secure FORTRAN", although it does not directly inherit FORTRAN's syntax [28].

Fortress's concurrency support is minimal, opting for a model based on automatic parallelisation. For example loops are implicitly executed in parallel, function and method calls can be evaluated in parallel. Atomic, mutually exclusive, regions are supported; however, unsafe concurrent manipulations are not precluded by default. Explicit futures are supported, and many of the implicit parallelism features use implicit futures.

2.5.13 Google Go

Type:	language
Origin:	industry (Google)
Year:	2009
Keywords:	*C-syntax*, *channels*, *CSP*, *data-oriented*

Google Go is a CSP-inspired language with a C-syntax. The developers advocate the use of concurrency for program structure, i.e. *expression* (2.1.1) [202]. It is the latest in a line of CSP-inspired languages which find their origins in the *Plan 9* operating system [203]. In particular Go's channels can be traced back to Newsqueak, a language designed for developing graphical user interfaces (an inherently concurrent problem) [201].

Go's unit of concurrent execution is called a *goroutine*. Channels provide synchronisation and communication between goroutines and can be synchronous or asynchronous. While channels are provided shared mutable data is also available and unsafe manipulation of the shared state is possible. Choice between channels and timers is possible via a *select* primitive. Thus Go supports process-oriented programming in additional to data-oriented style concurrency.

2.5.14 GPGPU: CUDA and OpenCL

Type:	library
Year:	2003
Keywords:	*C-syntax*, *data-oriented*, *vectorising*

CUDA and OpenCL are two extensions to the C programming language to enable so called "General Purpose computing on Graphics Processing Units" or GPGPU. In essence both extensions permit the general programming of specialised highly parallel vector processors designed for handling floating point numbers. The use of a mixture of processor types, for example a GPU together with a general purpose processor, is often called *manycore*.

At its simplest the programmer implements a *kernel* which will execute in parallel on the GPGPU hardware. The kernel performs computation on a block of data modifying it or producing separate output. The runtime system replicates the kernel across the available hardware resources, perhaps creating hundreds of instances. The kernels process data which has to be partitioned a priori by the programmer.

Kernels can contain barrier synchronisation operations; however, these and branching within a kernel greatly reduce performance. The kernels are executed lock-step on a massive vectorised processor hence if one kernel diverges from the execution path the other kernels it shares an execution unit with must stall. Fundamentally this makes GPGPU programming much more suited to well-defined processor intensive tasks rather than dynamic or responsive applications. GPGPU maps well to the Stream programming paradigm [62].

2.5.15 Haskell

Type:	language
Origin:	academia
Year:	1990
Popularity:	33
Keywords:	*data-oriented*, *functional*, *futures*, *lazy*

Haskell is a pure and lazy functional language. Concurrency in Haskell focuses on deterministic and often implicit parallel graph reduction, a safe form of data-oriented concurrency. This relies heavily on Haskell being a lazy functional language. All values in Haskell are internally represented by primitives called *thunks*. A thunk is a suspended closure which on activation computes a value to replace the thunk. These are equivalent to futures which are resolved as needed. Significant work has been done on automatic parallelisation of these potential futures [124, 170].

While non-deterministic interaction between concurrent components is not an intended programming model for Haskell, it does support concurrency primitives. Haskell's *MVar* primitive behaves as a one-place buffered channel (2.3.2.4) [123]. Additionally work has been done on implementing CSP-inspired computation in Haskell [60]. These mean that in principle Haskell can support process-oriented programming.

2.5.16 Intel Thread Building Blocks

Type:	library
Origin:	industry (Intel)
Keywords:	*data-oriented*, *vectorisation*

Intel Thread Building Blocks is a C++ library which provides templates for breaking programs structured around shared data and loops into *tasks*. An efficient task scheduler executes these tasks using a pool of threads suitable for the host system. Tasks are more fine-grained than direct POSIX threads and hence the resulting software has a higher degree of internal concurrency. The programmer is permitted to think at a higher level than threads, focusing on computation to be done rather than partitioning computation accross threads and maintaining synchronisation.

Tasks are assumed to contain non-blocking computations which have implicit synchronisation and dependencies as defined by the loop structure which created the task. For example the `parallel_for` construct executes like a traditional for-loop, but creates one task per element. In order to avoid excessive synchronisation overheads, tasks are automatically coalesced in a number of constructs. The model is broadly similar to that of Unified Parallel C (2.5.28), and as vectorisation is similar to GPGPU (2.5.14).

2.5.17 Java

Type:	language
Origin:	industry (Sun)
Year:	1995
Popularity:	2
Keywords:	*C-syntax*, *data-oriented*, *monitors*, *OO*

The Java programming language is broadly similar to C++ (2.5.6) in motivation and design. It takes a C syntax and adds an object-oriented programming model and standard library [117]. Java's object-oriented model is simpler than C++'s and its standard library significantly more extensive. However the critical difference between the two is that Java uses a managed runtime environment, which allows programs written in Java to run on any computer with the appropriate runtime environment without being specifically compiled for that computer. The managed runtime environment also provides automatic memory management, removing memory allocation and deallocation concerns from the programmer. Essentially the runtime environment or runtime system is a *virtual machine* (2.1) which is simulated by the host computer. The Java virtual machine is now used by many more languages than Java, notably Scala (2.5.26) and Clojure (2.5.9).

Java incorporates a basic data-oriented concurrency model by design. Every object has an associated monitor and blocks of code can explicitly synchronise using this monitor to provide mutual exclusion. Like C++, Java defines a memory model describing the interactions of concurrent modifications to shared data, although its initial specification was flawed and required significant revision [206, 168]. The standard library has grown significantly over time and now includes many concurrency primitives, such as barriers, futures and message queues which can be used as synchronous channels (2.3.2.4). Choice over these primitives is not supported, athough message queues and futures can be polled for readiness.

2.5.18 Manticore

Type:	language
Origin:	academia
Year:	2007
Keywords:	*channels*, *functional*, *messages*

A successor language to Concurrent ML (2.5.10), Manticore attempts to address heterogeneous parallelism [108]. Heterogeneous parallelism describes systems composed of multiple processors with different capabilities, for example a computer with a general purpose processor and a graphics co-processor (see GPGPU 2.5.14). Essentially Manticore incorporates all primitives of Concurrent ML, channels, messages and choice, and thus also supports process-oriented programming. It adds multiprocessor execution and data oriented vector operations. In order to test different scheduling models various granularities of scheduling are supported including *processes*, *threads* and *fibers* (2.3.1).

2.5.19 Message Passing Interface

Type:	interface
Origin:	academia
Year:	1992
Keywords:	*channels*, *messages*

Message Passing Interface (MPI) is a standardised system for the exchange of messages between distributed computers [236]. Its goals are broadly similar to the Parallel Virtual Machine (2.5.23). Support for MPI has been implemented in a broad range of languages and thanks to standardisation these implementations can (in principle) interoperate.

MPI provides operations for starting concurrent processes; technically one is created for each physical processor in the computer system. These processes are considered isolated and do not share memory. Processes create *windows* into which data can be written with a *put* operation or retrieved with a *get* operation. These windows are in a sense buffered channels (2.3.2.4). In addition to these primitives, *collective* operations are supported for broadcasting or scattering data to a number of processes, and/or gathering data responses. MPI provides a coarse-grain form of process-oriented programming, although synchronisation support is limited.

2.5.20 Objective-C

Type:	language
Origin:	industry (Apple)
Year:	1983
Popularity:	3
Keywords:	*C-syntax*, *messages*, *OO*

Objective-C adds object-oriented style programming to C based on Smalltalk's messaging style (2.5.27). This message model, where objects are used by sending them messages, contrasts with C++ (2.5.6) which also adds object-oriented programming to C and was developed at broadly the same time as Objective-C. As a consequence of this difference in design, Objective-C permits dynamic typing and asynchronous method invocation. Like Smalltalk this makes Objective-C an Actor model language, although objects are not explicitly concurrent entities.

Objective-C relies heavily on an event-driven style of programming (2.2.3), with one or more threads dispatching events to objects by invoking their methods. There is a *main thread* where the majority of events are dispatched. As the dispatch loop serialises concurrent events, data-oriented concurrency errors can be avoided unless an event is explicitly assigned to another thread, in which case it should be designed to handle concurrency. The disadvantage of this approach is that long running methods can block the dispatch of new events.

More recently Objective-C has added support for *blocks* which are closures (2.3.1.3). These can be placed on queues from where they are executed on available processors by a system called Grand Central Dispatch. Objective-C has seen a recent surge in popularity as it supports development for Apple's popular mobile phone and tablet devices.

2.5.21 occam

Type:	language
Origin:	industry
Year:	1983
Keywords:	*channels*, *CSP-inspired*, *process*

The occam programming language was developed to directly support the programming of the Transputer processor [172]. The Transputer had a hardware scheduler which supported inter-process communication via synchronous channels. Critically the Transputer also had four external communication links which were addressed by software in the same way as internal communication channels, providing an almost seamless mechanism to distribute a concurrent program accross multiple physical processors [53, 138].

Channels in occam have two endpoints and are unidirectional. The sender writes to one end of the channel, the receiver reads from the other. For the communication to complete both parties must be present to complete the rendevous. A process may optionally wait for input on a number of channels concurrently, providing choice. Choice over output is not supported.

Due to the low overheads associated with process creation and context switching provided by the hardware scheduler, the occam language permits concurrency on a line-by-line basis at the source code level. The associated design methodology is that scenarios typically handled by explicit programmer serialisation of a problem or use of polling be instead solved by use of concurrent program elements.

As a programming language, occam closely matched the Transputer hardware in its feature set and could be described as a high level assembly language for the Transputer. However in addition to the hardware access provided by occam, it implemented strict control of resources to prevent race conditions. Specifically all unsafe uses of shared memory are disallowed at compile time and channel ends can only be held by a single writer and reader at a given time. As such the semantics of the language closely model those of CSP by design (2.2.2), and occam supports process-oriented programming.

An extension called occam-pi adds support for the mobility of data and channel ends based on the pi-calculus [257]. Prior to this extension the program network would need to be computable at compile time. With occam-pi any number of processes can be created at runtime and connections between these arbritarily established and changed.

2.5.22 OpenMP

Type:	interface
Origin:	industry
Year:	1997
Keywords:	*barriers*, *data-oriented*

OpenMP is a language agnostic interface to shared-memory multiprocessing facilities [19]. It was developed by a consortium of commercial organisations, but the specification is maintained not for profit. Due to the Fortran interface, OpenMP is often used for parallelising scientific computations which are written in Fortran [69, 93].

The programmer annotates existing programs with OpenMP pragmas, allowing for parallelising existing code. The primary programming style is loop parallelism similar to Intel Thread Building Blocks (2.5.16). Annotated loop statements are executed in parallel such that the system runtime then divides the loop work between available processors. Additional annotations then modify the visibility and access patterns of data between parallel computations and add synchronisation points, such as *barriers*. OpenMP's simplified concurrency model aids conversion of existing program code compared to other methods such as POSIX threads (2.5.24), although the runtime may use these to implement the annotations. The major limitation of this approach is that it assumes a shared-memory computer systems. Additionally parallelising of existing code without static checking can lead to unforeseen errors [241].

2.5.23 Parallel Virtual Machine

Type:	library
Origin:	academia
Year:	1989
Keywords:	*messages*

The Parallel Virtual Machine (PVM) was developed to allow the programming of a number of separate distributed and heterogeneous computers as a single *virtual* machine [112]. At its core PVM provides operations for starting tasks on remote computers and communicating with them via messages. This structure acknowledges the architecture of the virtual machine is distributed and provides tools for moving data and routing messages. PVM was suceeded by MPI (2.5.19).

2.5.24 POSIX Threads

Type:	interface
Year:	1993
Keywords:	*data-oriented*, *monitors*, *semaphores*

POSIX threads are the most common means to access concurrent execution of program elements. All common operating systems support POSIX threads and likewise access to them is widely integrated in programming languages. Each POSIX thread created by a program executes concurrently with all other threads in the program and shares memory space with them. Threads are operating system primitives and may execute in parallel if the operating system schedules them as such. Access to memory by concurrent threads is not mediated and without use of additional POSIX primitives or suitable hardware instructions it is not possible to avoid race hazards between threads.

The programmer can mediate concurrent memory access using POSIX primitives, the most common of which are mutual exclusion locks and condition variables (2.3.2.1). A mutual exclusion lock can only be held by one thread at a time, other threads attempting to take the lock must wait for it to be released (or poll it as appropriate). This permits the implementation of critical regions in which only one thread modifies an area of memory. A condition variable allows the signalling of one or more threads which are waiting for the signal. The overheads of POSIX threads and associated primitives are heavily dependent on the host operating system's implementation. Further to this as POSIX only provides low-level primitives there is no standard design methodology for POSIX threads.

2.5.25 Rust

Type:	language
Origin:	community (Mozilla)
Year:	2010
Keywords:	*C-syntax*, *channels*

The Rust language, in development by the Mozilla Foundation, uses a C++ inspired language syntax (2.5.6) and provides explicit concurrent programming [23]. A concurrent component is a task, which is a form of coroutine (2.3.1.3). Unsafe sharing is forbidden by design, i.e. there is no mutable shared data. Sharing is facilited by communication through synchronous and asynchronous channels. Each task has *ports* on which it receives messages from channels connected to that port. Choice between ports is provided by a *select* primitive. As such Rust provides a strong process-oriented programming model.

2.5.26 Scala

Type:	language
Origin:	academia
Year:	2003
Popularity:	35
Keywords:	*actor model*, *channels*, *functional*, *futures*, *mailboxes*, *OO*

Scala was designed as a successor to Java (2.5.17) integrating functional programming features. This is enabled by seamless integration between Java and Scala code made possible by the use of the same virtual machine runtime enviroment (JVM) [229]. Scala provides a concurrency model based on Actor model semantics with asynchronous messages and mailboxes [195]. The sender address is automatically embedded in messages. This enables a mechanism for simulating synchronous messaging and for forwarding messages impersonating the original sender. Buffered channels are provided using synchronous messaging with an actor emulating the channel, no direct communication between actors becomes a special case of channel communication. This allows protocols to be defined between senders and receivers.

2.5.26.1 Akka

As of version Scala 2.10.0 released in January 2013, Scala's Actor model implementation has been migrated to an external library called *Akka*. Akka's implementation is also usable from Java (2.5.17), and supports distributed communication between separate programs which may be on separate computers.

2.5.27 Smalltalk

Type:	language
Origin:	industry (Bell Labs)
Year:	1973
Popularity:	37
Keywords:	*actor model*, *messages*, *OO*

Smalltalk was one of the first object-oriented programming languages and a direct descendent of Simula 67 [150]. In Smalltalk everything is an object, including primitive types such as integers. All operations on objects (method invocations) occur semantically and syntactically as the passing of messages between objects. For example the addition of two numbers is the sending of a addition message to the first operand with the second operand as the message body, the result being returned as a message.

Objects are all independent entities (instances of classes) with their own memory. While objects are conceptually concurrent entities under normal circumstances execution of one object is suspended when it sends a message to another. Multiple threads of execution can be created and are scheduled by the language runtime, allowing for concurrent execution of objects and their method invocations [238]. Smalltalk was pivotal in the development the Actor model (2.2.2) [74].

2.5.28 Unified Parallel C

Type:	language
Origin:	academia
Year:	1999
Keywords:	*barriers, data-oriented, messages, PGAS*

Unified Parallel C (UPC) adds explicit support for distributed shared memory or partitioned global address space systems to C [81]. UPC adds keywords for distributing operations across all available computation elements in the computer system and allocating memory or globally with respect to the location of a computation. Accesses to global memory are implicitly converted to messaging requests between computation elements. In order to synchronise distributed computations barriers and locks are added. Summarily, UPC adds the minimum level of support required run a single C program on a large distributed computer system. The goal is the same as MPI (2.5.19), but using a data-oriented rather than message-passing model.

2.5.29 X10

Type:	language
Origin:	industry (IBM)
Year:	2004
Keywords:	*barriers, data-oriented, futures, monitors, OO, PGAS*

X10 is an explicit parallel programming language, which similar to Chapel (2.5.7) has been developed as part of IBM's involvement in DARPA's High Productivity Computing Systems program [70, 4]. A Java-like (2.5.17) syntax is used with a partitioned global address space model (2.2.1), which allows an object-oriented style of programming with globally addressable objects. This is a very intentional design intended to allow easy transformation of existing Java code to X10.

X10's primary concurrency mechanism is an *async* statement which begins computation concurrent to the running computation. This is similar to Cilk (2.5.8). These computations can be distributed between available computation sites or *places*. Atomic statements which serialise access to data are available: these may also be conditional which makes them essentially *monitors* (2.3.2.1) without low-level access. In addition to these mechanisms *futures* are also provided. Finally, *barriers* (2.3.2.2) and an extended form called *clocks* are provided for synchronising concurrent computations. These mechanisms imply locking, but low-level locking is not available to the program code so as to avoid data-oriented deadlock issues.

2.5.30 XMOS XC

Type:	language
Origin:	industry (XMOS)
Year:	2005
Keywords:	*C-syntax*, *channels*

XMOS XC is a version of C which adds support for processes and channels, in order to support XMOS's xCORE processors [174, 252]. This mirrors the development of occam (2.5.21) for the Transputer processor (2.6.2). Processes in XC are mapped directly to hardware computation elements meaning the programmer is limited at design time to the number of threads available in processor (a multiple of the number of cores). This is a contrast from occam which was not limited by the number of processors; however, this restriction allows the processor to guarantee the response time of any given concurrent process, a critical concern in real-time systems.

The XC compiler disallows unsafe data sharing between concurrent processes. Synchronous and asynchronous channels can be used to support data sharing between processes. Additionally *ports* are provided for communication between processes and external components. A port can have a communication rate with which a process can synchronise. Choice over channels, ports and timer events is provided.

2.6 Concurrency Support in Hardware

Previous sections have broadly reviewed support for concurrency mechanisms in programming languages. This section describes the development of support for these mechanism in hardware.

In section 2.1.1, I described how a computer's operating system needs to address concurrency concerns related to the concurrent operation of hardware. However the description given assumes only a single processor which manipulates the computer's state. In the early 1960's computers with multiple processors began to emerge, for example Burroughs D825 in 1962 which could have up to four *computer modules* [32]. Multiple processors are desirable to increase the computational capacity of the computer and provide *fault tolerance*, whereby the computer can continue to operate with reduced capacity in the event of component failure. These systems were *multiprogrammed*: a separate program was run on each computer and there was no direct interaction between these programs. Only the operating system caused interaction between processors in order to assign processors and other resources to programs.

Another early commercial multiprocessor system the UNIVAC 1108 in 1966 added an explicit *test and set* instruction [237]. This allowed the programmer to atomically test and set a memory location in the same cycle. If the memory location was already set then the instruction would cause an interrupt and invoke the computers operating system to intervene. This mechanism allowed mutual exclusion of critical regions.

Many early computer systems used asymmetric multiprocessing. In an asymmetric system not all processors run at the same speed or have the same access to system resources. Input and output devices might be connected to a main processor which is responsible for managing the system and moving data, worker processors are then connected to the system's memory. Any output from a worker processor must go via the main processor. Early asymmetric multiprocessing was primarily motivated by cost, for example slower cheaper worker processors could be added to an existing system structure to increase capacity cost effectively [85].

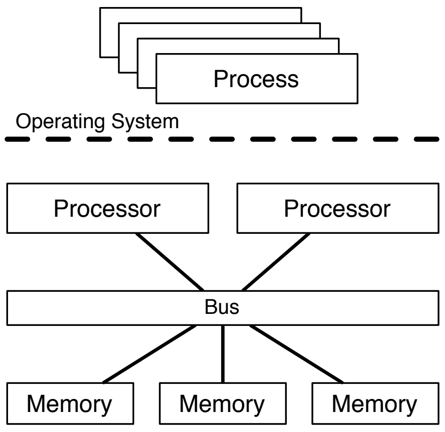


Figure 12: Overview of a symmetric multiprocessor architecture. Processes are multiplexed onto processors by the operating system. Memory banks are accessed by processors over a shared bus.

The alternative to asymmetric multiprocessing is symmetric multiprocessing (*SMP*). All processors run at the same speed and may use the same clock signal synchronising their communication with shared memory and other devices. In such a setup all processors have equal access to system resources and programs can execute on any part of the computer with the same performance characteristics. A symmetric design assumes a shared memory space, as shown in figure 12.

In the 1970s computer networks began to emerge, the most notable being ARPAnet, a forerunner to the modern Internet. These networks provide another view of asymmetric multiprocessing where a larger computer system can be built from many smaller computers without a shared memory space. Up to this point data in a multiprocessor system was transferred through shared memory, but without shared memory explicit communication via messages was required.

By the early 1980s computer systems can be segregated into three tiers based on scale:

- Micro-computers: single processor machines which fit on a desk and have a single user.

- Mainframes: servers and workstations, larger installations serving many users or users with special requirements. These essentially maintain the structure of a single SMP computer.
- Super-computers: systems composed of many independent computers, used predominately for scientific computing.

The micro-computer, or *personal computer* tier was heavily influenced by the Apple II in 1977 and the IBM PC (Model 5150) released in 1981.

Through the 1980s and early 1990s personal computer makers grow in size and strength, micro-computer architecture systems begin to be used for tasks previously done by larger mainframe computers. However at the same time super-computers continue to grow in size exponentially. A new tier of *embedded computers* emerges, devices which are not general-purpose computers, but contain a computer to provide or enhance functionality. Classic examples of embedded computers are video recorders, washing machines and computer games consoles.

By the mid-90s five main processor types are in use, each focusing on a particular tier:

- *DEC Alpha*, used in servers;
- *IBM POWER*, used in mainframes, specialist computer systems, and a derivative called *PowerPC* used in personal computers, particularly those built by Apple;
- *Intel x86*, used in personal computers;
- *MIPS*, used in embedded systems and SGI super-computers and workstations;
- *Sun SPARC*, used in servers from Sun and super-computers from Cray.

This field then shrinks over the next decade and up to the present day. DEC Alpha ceases to be developed and is sold to Intel. Cray and SGI merge before Cray sells parts of its business to Intel. Sun ceases development on SPARC and switches to Intel architecture before being bought by Oracle. Apple switches from PowerPC to Intel x86 for its personal computers. MIPS sees a significant decline in use. A range of architectures using IBM POWER are unified under the more general Power instruction set architecture (Power ISA). The x86 architecture is licensed by other processor manufacturers, most notably AMD, who extend it with 64-bit support. However, strengthened by a growing mobile phone market, the ARM architecture has grown to significant prominence. Additionally, special purpose graphics co-processors based on vectorising (SIMD) have gained general programming capability, *GPGPU* (2.5.14), and find use in scientific computation.

Essentially this leaves four major processor types in use today: ARM, Power, x86 and GPU. All of these have gained multiprocessor or multicore support through the direction of recent changes in hardware development (2.6). Evolving from single to multi-processor architectures all of these processor types have a shared memory model. However the behaviour of this shared memory varies between processor types.

All modern processors contain *cache* memory. This cache maintains copies of data stored in main memory, but is much faster to access. Faster memory (SRAM) used to build cache is more expensive to produce and requires more energy to operate, so only small quantities are provided. Cache is required because processors operate significantly faster than main memory (built from DRAM), or rather it is not cost-effective to have a main memory (built from SRAM) that operates at the same speed as the processor. Main memory can also be seen as a cache for persistent storage on disks.

Cache forms a hierarchy, with level one or *L1 cache* closest to the processor core. Assuming a normalised size of 1 for the $L1$ cache, then main memory is in the order of 10^5 times the size of $L1$. Typically a second level of cache, *L2 cache*, will be added to the processor to further smooth performance differences, and this is 10^2 times the size of $L1$ cache. The L2 cache may be shared between cores, but if it is not then often a further *L3 cache* is added which is shared between cores, this is around 10 times the size of $L2$, so 10^3 times the $L1$.

Data within a processor cache memory is organised in *cache-lines* [99]. These are 64 bytes in size on x86. When data is loaded from and written to main memory, or between caches, it is moved in cache-line sized units. This is to mitigate the latency effects of memory accesses, i.e. the round trip time to access memory is the same up to a particular size. Therefore we should move the maximum amount of data possible to avoid extra trips. To analogise, if it takes the same minimum amount of time for a letter to travel through the postal system as a parcel, then we can send or receive 100 letters (in a parcel) from the same places in the same time as one letter.

Cache has two implications in a multiprocessor system.

1. It reduces load on the shared main memory. This is important as the main memory *bandwidth*, the amount of data that can be transferred into or out of main memory, will be shared.

2. Aliasing and synchronisation problems can occur because multiple copies of the same data can exist in the system. Cores working on the same data must maintain *cache coherence*.

Cache coherence is maintained by inter-processor communication. Simplistically, when one processor updates a cache-line in main memory, it must *invalidate* any copies of that cache-line in *all* other processor caches [143]. Hence when another processor reads the data it retrieves an updated copy from the main memory. Such invalidation is however not immediate. If one processor wishes to modify a cache-line which has been modified and cached by another processor it must request ownership from that processor. This can involve the cache-line being written back to main memory before being loaded by the requester[7]. Compared to simply loading data from memory this synchronisation is a time consuming operation. Worse if the original owner wishes to modify the cache-line again, the whole process must be repeated. This effect is called *cache ping-pong* as the cache-line moves back and forth like a ping-pong ball. Cache ping-pong occurs when two processors are working closely on a shared piece of data [99], or on data which is not intended to be shared but is on the same cache-line, *false sharing*.

[7]Modern processor architectures can transfer cache-lines between directly between processor caches without writing back to main memory.

Multiprocessor memory access is further complicated by the superscalar behaviour of modern processors such as read and write buffers. Write buffers in the processor mean it can take many steps for a given update to memory to reach the cache let alone the main memory. Within the same processor reads will load values from the buffer, but the contents of this buffer are not visible to other processors. Critically the processor might reorder writes in the buffer to improve performance (sequential writes to memory are faster than unordered writes) [99]. The order of memory operations (reads and writes) performed by the program and the order of operations observed by other processors (and hence programs) in the system may not be consistent [169]. This matters because synchronisation between concurrent components may depend on the order of operations [157]. Imagine that component A writes memory at location X before modifying location Y to indicate data is ready in X. Component B tests location Y to see if data is ready before reading it from location X. For this synchronisation to work the changes to X must become visible before or at the same time as Y. This order of events often does not hold true in a modern multiprocessor system [199].

Memory order is one of the large differences between the major processor types. Intel x86 provides a *total store order* model, such that writes to a memory location by one processor are seen in the same order by other processors [143, 199]. This model guarantees that writes become visible to all processors at the same time. Power and ARM processors do not provide the same guarantees. In the case of Power this is for mostly historic and performance reasons. For ARM this is to reduce complexity and power consumption from the additional hardware required to implement total store order. These architectures can arbitrarily reorder reads and writes to memory, including speculatively reading data, and do not guarantee that writes become visible to all other processors at the same time.

2.6.1 Synchronisation

Within a shared memory multiprocessor system, concurrency support is provided by machine instructions to synchronise the state of a given processor and the shared memory. These instructions allow the programmer or programming language system to control the behaviour of the processor with respect to shared memory. This is necessary for the reproducibility of behaviour [77].

In a high-level language it is desirable, even expected, that synchronisation operations be automatic and that low-level processor details be removed, abstracted, from the concerns of the programmer. Thus programming language and runtime system implementers must use these operations to ensure consistent high-level behaviour. It is worth noting that despite the desirability of high-level behaviour, data-oriented concurrency in programming languages often unavoidably exposes low-level synchronisation details to the programmer [168, 20]. This is through either incomplete specification of high-level behaviour or the complexity of ensuring desirable high-level behaviours in the data-oriented paradigm (2.7).

Synchronisation instructions can be divided into two categories:

- Barriers, which enforce the order of operations on shared memory,
- Atomic operations, which modify shared memory in a consistent manner irrespective of concurrent reads and writes.

A barrier instruction prevents reads or writes to memory being reordered past the barrier. This controls the pipelining and superscalar behaviour of modern processors. Additionally the barrier may synchronise the state of cache in the processor with main memory.

On x86 barriers are provided by *mfence* (memory fence), *lfence* (load fence) and *sfence* (store fence) instructions [143]. Cached data can also be invalidated with a *clflush* (cache-line flush) instruction which forces a specific cache-line to be written back to main memory if it has been modified. On Power *sync* and *lwsync* (light-weight sync) instructions synchronise processor state and order with shared memory [145]. On ARM *dmb* (data memory barrier) and *dsb* (data synchronisation barrier) instructions flush processor pipelines synchronising the order of instructions and the order of memory operations visible by other processors [36].

An atomic operation allows one processor to exclusively modify a memory location or alternatively detect if concurrent access has occurred on a location. Historically x86 supported a hardware connection (shared wire) to lock all shared memory for exclusive access. This meant that most single instructions could be made atomic by prefixing them with a *lock* operation. While exclusive access to memory is no longer managed with a hardware connection, reducing contention, x86 still supports the same set of atomic memory operations. This includes instructions which read and modify the memory in the same instruction, for example *inc* (increment) can be used with a *lock* prefix to atomically increase the value of a memory location by one. The most widely used atomic sequence is a *cmpxchg* (compare and exchange) instruction. This tests the value of a memory location M against a value X, if M has the value of X, then M is set to a new value Y. In the event that M is not X then M is not modified. In either case the success or failure is recorded and can be used later. This is known widely as *compare and swap* or *CAS*.

CAS can be used to implement other atomic operations. For example increment can be achieved by reading a memory location, then applying CAS with the value and the incremented value. If the CAS operation fails then all steps must be repeated until the CAS succeeds. Critically it can take an unbounded number of attempts to achieve success due to the unpredictable nature of the interfaces between processors.

Power and ARM use an alternate mechanism to CAS. A *load-linked* (Power) or *load-exclusive* (ARM) instruction loads a value from memory and records a reservation for that memory location. After an arbitrary sequence of instructions a *store-conditional* (Power) or *store-exclusive* (ARM) instruction is used to write to the same memory location. If the reservation is still valid these instructions succeed and update memory, otherwise if another processor has modified the memory then the reservation is no longer valid and memory is not changed. This mechanism is referred to as *LL/SC*. Broadly

Year	Architecture	HW CPU	HW Memory	LW CPU	LW Memory
2005	IBM POWER4	57.20	15.26	9.20	2.45
2006	Intel Core	2.96	1.06	2.57	0.92
2009	Intel Nehalem	8.26	3.88	1.57	0.74
2011	Intel Sandybridge	10.17	3.99	1.67	0.65
2011	AMD Bulldozer	21.82	13.85	21.80	13.84

Table 1: Cost of memory barrier operations in CPU and memory bus clock cycles on recent processor architectures. *HW* means heavy-weight barrier, *LW* means a light-weight barrier.

Year	Architecture	CPU	Memory
2002	Intel NetBurst	144.24	15.99
2006	Intel Core	43.92	12.21
2009	Intel Nehalem	16.34	7.68
2011	Intel Sandybridge	19.04	7.46
2011	AMD Bulldozer	49.98	31.73

Table 2: Cost of *compare and swap* (CAS) operations in cycles on recent processor architectures in CPU and memory bus clock cycles.

speaking LL/SC has more expressive power than CAS [128], in particular LL/SC can detect when a value has been changed but holds the same value after changes, whereas a CAS will always succeed if the memory matches the comparison value.

Synchronisation operations have a performance penalty as they partially or fully negate buffer and cache effects forcing the processor to operate at the speed of (shared) memory. Interaction between processors may be required, which will occur at the speed of their respective interconnects. Table 1 shows the cost of barrier operations on recent processor architectures. Results are shown for two types of barrier, heavy-weight and light-weight, on x86 these correspond to *mfence* and *sfence* respectively. Heavily superscalar architectures such as Power suffer a significant performance penalty from barriers. This is seen in IBM POWER4 data and successive generations of Intel processors. Light-weight barriers suffer a much lower performance penalty and it is clear that Intel is optimising their performance on successive processor generations, while AMD clearly does not support light-weight barriers. In general light-weight barriers should be sufficient for most algorithms [169].

Table 2 shows the cost of CAS operations on recent processor architectures. Over successive generations the cost of CAS has decreased irrespective of processor and memory speed increases. The inconsistent result is again AMD's x86 architecture, this may be because it supports significantly more processor cores (64 in the generation tested) and synchronisation on an architecture of this scale is prohibitive. It is worth noting that CAS operations contain an implicit barrier. If used correctly this means there is no need to pay twice for synchronisation in a total store order system.

2.6.1.1 Measurement Methodology

The values in table 1 were computed by reading or writing ascending words of memory interleaved with the associated barrier instruction. This is done for 2^{24} machine words and repeated 20 times. The mean execution time is computed from the time taken for the above steps. A second test run taken at a later time is used to validate the first result. The result is then normalised by the processor and memory clock frequency of the specific test hardware. For table 2 the same methodology is used; however, in place of read, write and barrier instructions a sequence of `LOCK` and `CMPXCHG` is used.

2.6.2 Interconnects

The alternative to a shared memory multiprocessor computer architecture is a distributed memory architecture with explicit inter-processor communication. Each processor has its own local memory and connections to the rest of the system. Because remote memory has different performance characteristics to local memory (local memory is faster), distributed memory architectures are have what is known as *non-uniform memory access* (*NUMA*). Figure 13 shows an example NUMA architecture.

Components on different processors interact by communication over the computer system's internal network. The processor can mediate access to its local memory so as to share it with the rest of the system providing the impression of a global shared memory equivalent in size to the sum of all local memories. This is the architecture

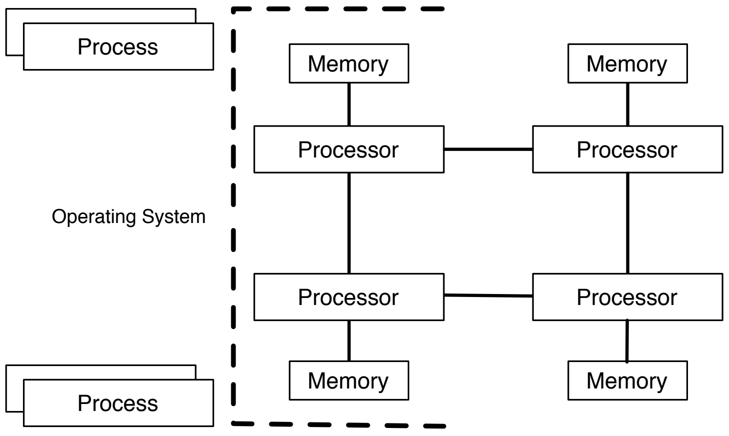


Figure 13: Overview of a non-uniform memory access (NUMA) multiprocessor architecture. Processors have local memory which is faster to access than remote memory.

partitioned global address space languages are designed to program (2.2.1.3). Such an architecture is used by most modern super-computers [121].

While general purpose networking technology can be used to interconnect components of a distributed computer system, such technologies are not designed for low-latency or fine-grain communication [207]. Thus special-purpose networks are used to directly connect computer processors together. Recent examples include Cray's Gemini interconnect [84] and Fujitsu's Tofu Interconnect [26]. Over the last 30 years the main activity of super-computer vendors has been to build interconnects for existing processors. This methodology has influenced processor design and modern processors integrate their own interconnect technology, such as Intel's QuickPath [141] or AMD's HyperTransport [80]. Thus while modern multiprocessor systems present a shared memory interface, they are in fact based on distributed memory architectures underpinned by message passing interconnects. Such a design is necessary as a single shared memory creates a point of contention, the performance of which degrades rapidly as the number of processors increases [118]. A side-effect of this distributed shared memory is that while globally accessible, different locations in memory have different performance characteristics [167].

It is worth noting that the one of the first processors to incorporate an interconnect technology designed explicitly for interprocessor communication was the INMOS Transputer [53]. The Transputer had four communication links which could allowed it to be inter-connected in grids and other complex configurations [176]. The process-oriented occam programming language (2.5.21) to which the technical work of thesis relates was designed primarily for the Transputer [172].

2.7 Analysis

This section discusses, connects and expands on trends emerging from previous sections, in particular sections 2.5 and 2.6. The goal is to understand the driving forces behind the development and use of programming languages, and relate these to process-oriented programming.

Despite previously identified issues (2.2.1) a data-oriented style of concurrent programming is popular. Given the post hoc incorporation of concurrent programming features into popular languages, we can postulate that it is the languages that are popular rather than the data-oriented paradigm. Data-oriented concurrency is used as it provides a method of extending existing software with concurrency support. Thus by understanding why programming languages are popular and the driving forces behind programming language development it is possible to establish the relevance of a process-oriented style to current trends in computer systems.

2.7.1 Popularity

In the previous section "Concurrency Support in Programming Languages" (2.5) it can be observed that the top five most popular programming languages have at most a minimal concurrency model. With perhaps the exception of Objective-C's event-driven UI model the concurrency support in these languages has been added recently (in comparison to when they were released). Additionally the programming style for concurrency

Language	TIOBE	Ohloh	langpop	Mean
C	1	2	1	1
Java	2	1	2	2
C++	4	3	3	3
PHP	6	6	4	5
Python	8	4	6	6
C#	5	8	7	7
JavaScript	10	5	5	7
Perl	9	9	8	9
Objective-C	3	10	15	9
Ruby	11	7	10	9
Visual Basic	7	11	12	10

Table 3: Languages ranked by popularity.

in these languages is predominately data-oriented (2.2.1), with few if any features to help prevent data-oriented programming hazards.

The TIOBE Index is the primary source of language popularity data used in this thesis [246]. It is derived from a mature and relatively stable collection method involving internet search engines and numbers of engineers, vendors and third-parties working with a given language. Table 3 shows a ranking of popular programming languages. Relative rankings for the top 11 languages from the TIOBE Index January 2013 were sampled from two other sources, Ohloh Monthly Commits [11] and langpop.com [8]. Ohloh tracks open source projects and their activity, and langpop.com tracks search engine results. These samples were combined with the TIOBE Index to derive a mean ranking for validation of the TIOBE Index itself. The results show the TIOBE Index to be broadly consistent with the mean rank, although a few anomalies are present. Fewer open source projects use Objective-C and Visual Basic. This explains the discrepancy between their TIOBE rank and other ranks. Additionally Objective-C's popularity is relatively recent (2.5.20) and langpop.com's data sources may be outdated. Conversely JavaScript is heavily used in open source projects causing a divergence between the mean and TIOBE rank.

Figure 14 shows the inter-relationship of the top 11 languages from the TIOBE Index. What is clear from this diagram is that all of the top five languages are one family, C and its decendants. From this it can be suggested that the concurrency model of the top five is an consequence of their heritage. Another observation is that all of these top five languages and the majority in the top 11 are industry led developments.

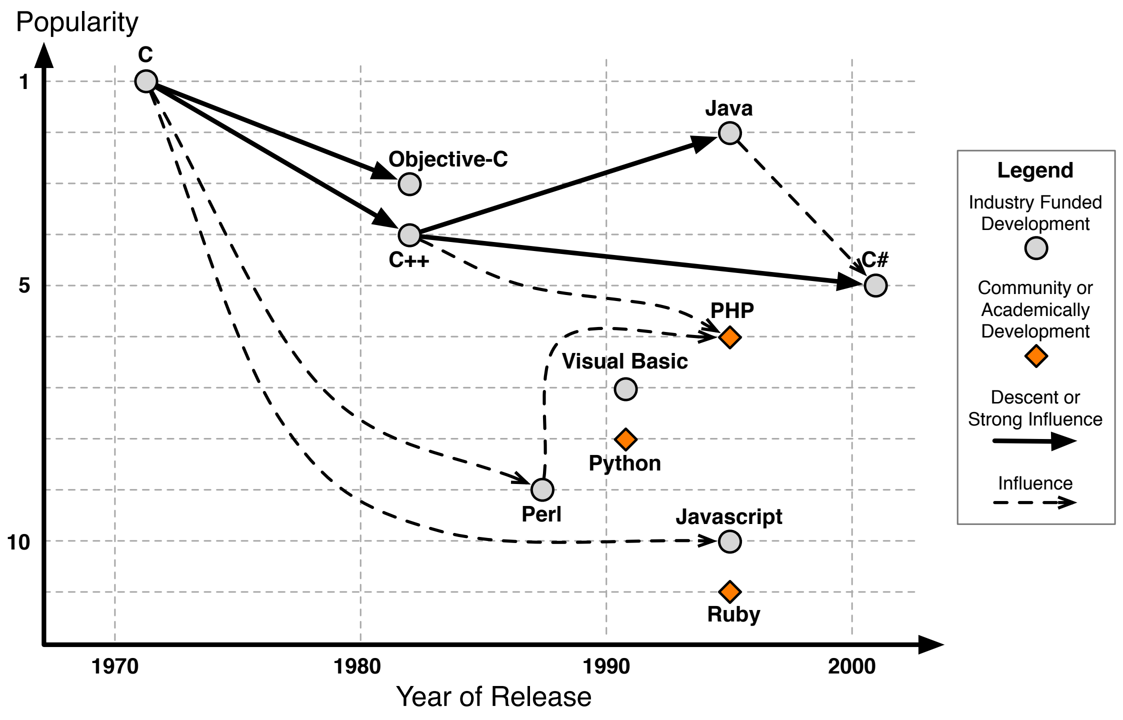


Figure 14: Language popularity and heritage, the relationships between popular programming languages.

This raises a general question; what makes a programming language popular? And, more specifically: **why are C and C-family languages as popular as they are?**

2.7.2 The Popularity of C

Popularity data suggests that C is not only the most popular language now, but that it has been for the last 20 years. There are two core features of C which gave it initial popularity:

- *Efficiency*, C converts directly to machine code and requires almost no runtime system. This means that C can be reasoned about and optimised with reference to a hardware model.

- *Portability*, the language and (optional) support libraries are relatively small. This means the development effort to produce a C compiler for a new platform is low.

These features contrast with other programming languages in the 1970s and 1980s which provided good abstraction from hardware concerns but required complex compilers and runtime systems, reducing their efficiency and portability. Both of these features are very intentional. In the late 1960s it was unclear whether any third generation programming language would be efficient enough to implement an operating system [208]. C was designed to implement the Unix operating system so had to be efficient, and it also had to be portable if Unix was to be made available on the vast range of computer systems available at the time.

A third reason has given C its enduring popularity: *ubiquity*. This can be attributed in part to the popularity of Unix. C provided the standard programmer interface to operating system functions. With the widespread adoption of Unix and Unix-like operating systems most programmers needed to have some understanding of C. Additionally, the market desire for Unix reinforced the concept that new computer systems should be released with a C compiler if they were to be of value [31]. In time programmers came to rely on the existence of a C compiler for any computer and operating system. Programmers building portable software, software designed to run on a range of computers, used C. This creates a cycle, as adding support for C allows (in principle) a large range of existing software to be used. The root of C's efficiency is *mechanical sympathy*, its programming model and the hardware model of computers are closely aligned (or were historically).

For the top 11 languages previously identified, each trades execution efficiency or cost of implementation (portability) for increased expression. For example, Java trades efficiency for a virtual machine with advanced optimisations such as just-in-time compilation. Combined with a large standard library this increases the portability of Java programs, which is reinforced by hiding machine details. Java has weak mechanical sympathy, but makes up for it with portability.

It is accurate to say that the designers and programmers of the first electronic computers were either one and the same or part of the same closely knit team. Software and hardware are largely indivisible. This trend continues with the first commodity computers sold by large equipment vendors such as IBM in the early 1960s. These early computers would be supplied with an assembler or perhaps a FORTRAN compiler. A team of programmers would then be hired to operate and program the computer to the bespoke needs of the purchaser. The term software begins to emerge in the mid-1960s and by 1969 software is a commodity with IBM charging separately for the software it supplies [208].

In the 1970s, while the software is becoming a separate commodity from hardware, hardware vendors continue to be the main developers of software. To the hardware industry, software is a means to sell hardware. For the majority of customers the hardware is only as good as the software it runs.

The introduction of microcomputers through the late 1970s and 1980s brings an increased need for software. Well-known software industry giants such as Microsoft and Oracle are founded. These companies provided software without being hardware vendors. This causes a separation of concerns. Figure 15 expands heavily on figures shown in section 2.1. The programmer, programming system designer and computer system designer are now separate entities. Each has their own *separate* and *distinct* view of how the computer works. The programmer uses a programming language which has an embedded *programming model*. The programming model will, to varying degrees, embed a *hardware model* based on the designer's understanding of specific or generic hardware. Critically the programmer also has their own hardware model which they have gained from experience of the programming language, education and other sources. Finally the hardware designer uses observations of programs and programming models to improve existing hardware or develop new hardware.

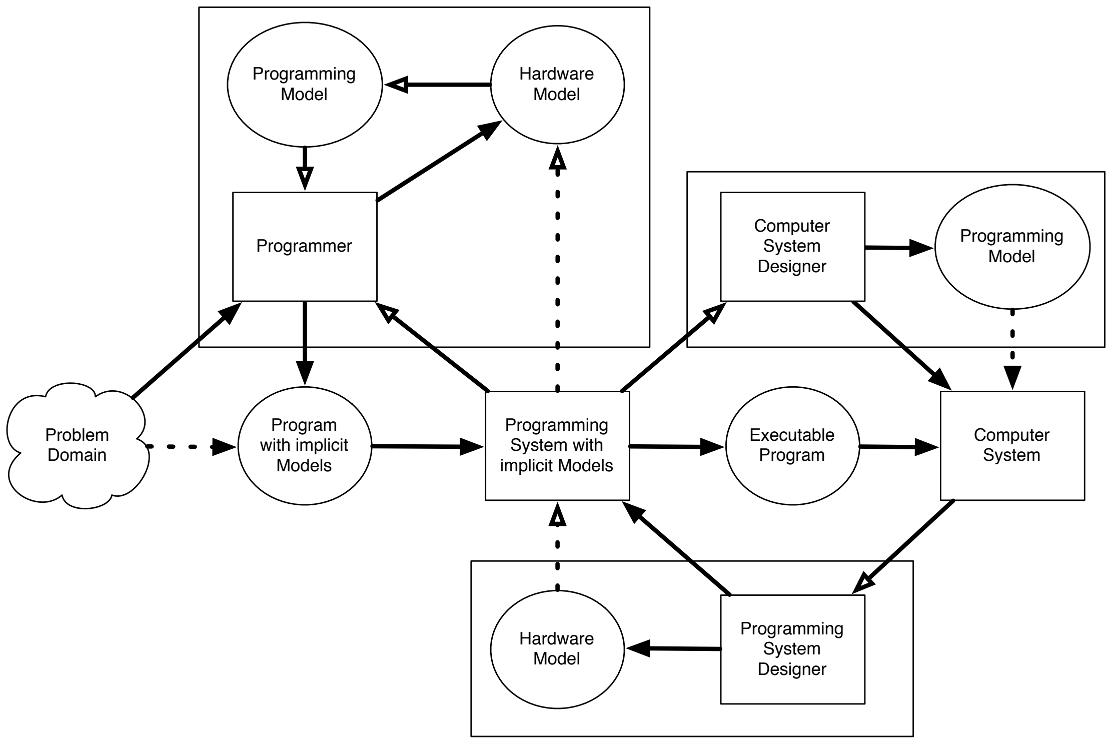


Figure 15: Overview of abstraction from problem domain to computer system highlighting that separate parties involved can have independent models of both the computer and programming systems.

Mechanical sympathy is the congruence of these separate models. Importantly this is not simply equivalence, but rather harmony. For example while a modern computer with a superscalar processor does not execute instructions in the sequence specified, its execution model is in agreement with the common hardware model used by programming language compilers. This shared understanding helps achieve optimal performance. If the models are not in agreement and the programming language does something not in keeping with common understand or the processor does not perform well for a common behaviour then this discord can reduce performance.

C's major flaw is that it sacrifices high-level expression to achieve mechanical sympathy. This obscures the problem model and increases program complexity (see 2.1.1).

2.7.3 Trends in Academia

In this context it is of value to analyse trends in academic research regarding computer science and more specifically programming languages. Figures 16, 17 and 18 show percentages of academic papers mentioning particular topics. Data is taken from all publications of ACM SIGPLAN Notices between 1970 and 2012 [1], 7739 papers in total. Aggregating a number of conferences and journals, ACM SIGPLAN Notices gives a good overview of *software related* issues in computer science research. Keywords were assigned to papers based on their existing bibliographic keywords and title text. Each figure shows the percentage of publications in a given year which mentioned a particular topic (composed of multiple keywords).

Figure 16 shows the rise and fall of interest in particular programming languages. The academic community appears to pick a new language every five to ten years. In recent times the peak in interest in a given language occurs about five to ten years after its release, although this has not always been true. Interest in FORTRAN rises up to a peak in 1977, the year FORTRAN-77 is released. A reason for the associated rise in ALGOL references is not clear. Ada peaks in the year of its release 1980. Smalltalk peaks in 1986 about five years after the release of Smalltalk-80, the first version available outside Xerox PARC. Interest in C family languages (by name) rises from the late 1970s (C was released in 1973) to a peak in 1992, nine years after the release of C++. This interest is maintained until the rise of Java approximately five year's after its release. Interest in Java has now declined, leaving room for a new hot language or topic.

Figure 17 shows interest in three main programming styles: objected-oriented, functional and logic programming. The spike in interest in object-oriented programming in 1986 coincides with the first OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) workshop on the topic. Within this figure languages such as Java and C++ are added to the count of objective-oriented papers, and functional and logic programming languages like Haskell and Prolog are aggregated into their respective topics. The purpose of this figure is to show that there is a steady increase in interest in alternatives paradigms, functional and logic programming, and a trend

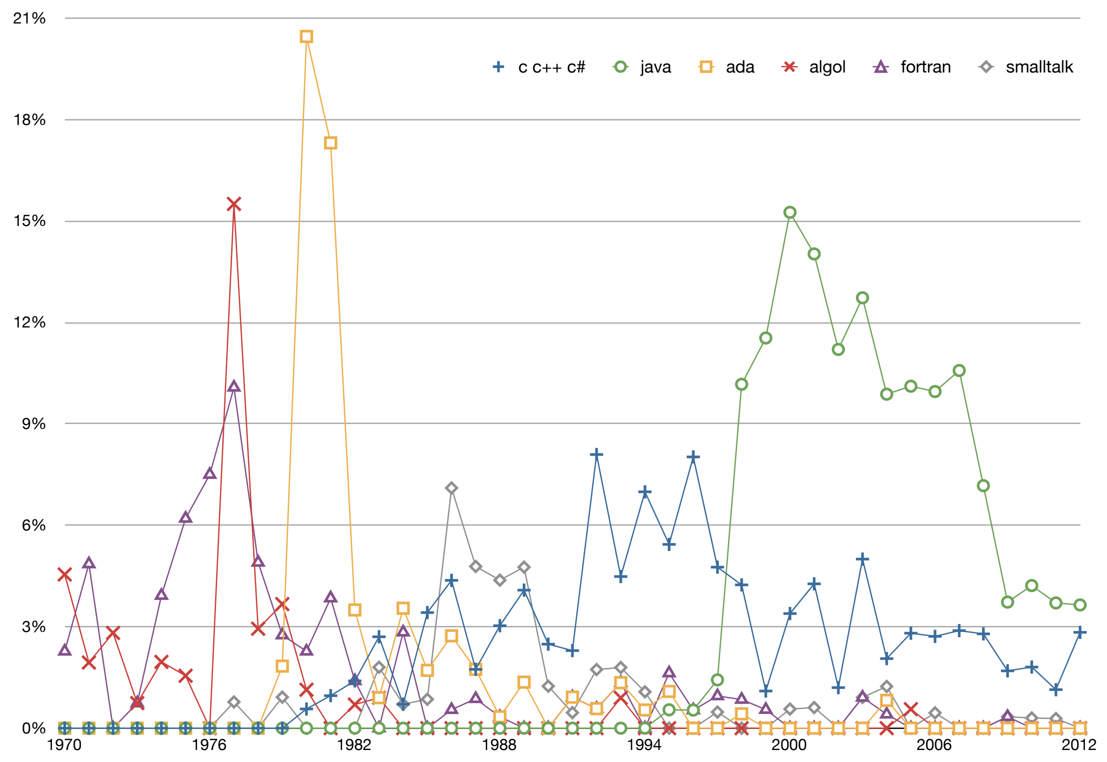


Figure 16: Percentage per year of ACM SIGPLAN Notice papers relating to particular programming languages.

away from object-oriented programming research. This contrasts with the TIOBE Index which shows interest in functional and logic programming falling year on year, with their present popularity levels approximately one third of their academic publication levels [246].

Figure 18 shows publications related to concurrency. Topics are separated into parallelism, concurrency, and multiprocessor related keywords. Related topics such as atomicity, scheduling, monitors, and semaphores are aggregated separately. Note that double counting is possible, this accounts for the common oscillations. The 1988 spike coincides with one of the first supercomputing conferences, SC88. From 2004, the year

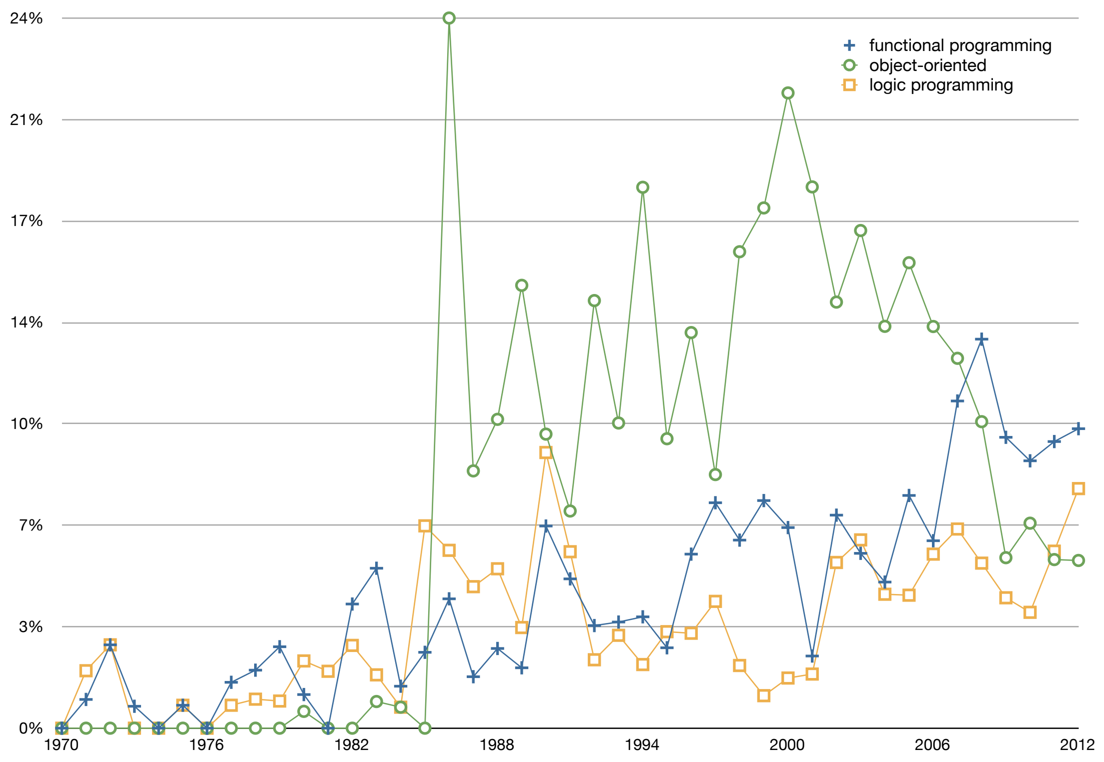


Figure 17: Percentage per year of ACM SIGPLAN Notice papers relating to particular programming paradigms.

of release of Intel's first dual-core processor there has been a gradual increase in concurrency and related topics publications. The noticeable jump in parallelism publications from 2007-2008 coincides with Nvidia's release of CUDA, its GPGPU programming language (2.5.14), although it appears there has been an ongoing interest in parallelism, probably buoyed by the supercomputing community. The key observation here is that academia is adopting hardware advances quicker than language changes.

From these figures it can be concluded that academia mirrors rather than sets trends. It is the commercial hardware and to a lesser extent software industry that defines direction. Academia is increasingly focused on alternative forms of expression. These forms of expression while powerful more often than not neglect mechanical sympathy.

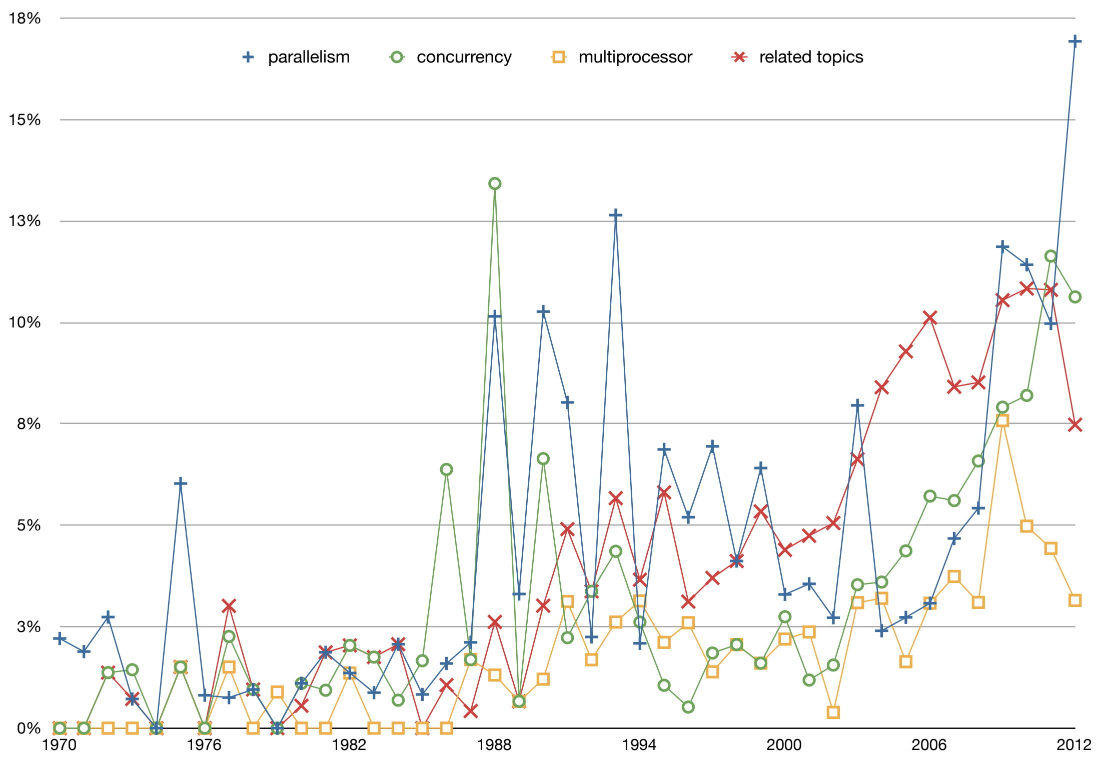


Figure 18: Percentage per year of ACM SIGPLAN Notice papers relating to concurrency topics.

2.7.4 Industrial Influence

Like any commercial enterprise the hardware and software industry has a clear goal in maintaining profitability through sales to existing and new customers. This is accomplished by creating new products which are in some way faster, more efficient to run, cheaper or have new and novel features. From a consumer standpoint the key metrics are *performance* and *price*. These ultimately drive the market, the hardware and software industry. For example, major rivals Intel and AMD can be typified as, Intel competes on speed while AMD competes on price.

Ultimately the shift to multicore and manycore is driven by the marketability of such performance. Performance which is consumed by "advances" in software engineering. *"The hope is that the progress in hardware will cure all software ills. However, a critical observer may observe that software manages to outgrow hardware in size and sluggishness."* [210].

It has been observed that software increases in complexity and reduces in efficiency as hardware becomes faster [264]. This is probably not a conscious behaviour, rather given increased resources there is less need to concern oneself with managing them, and overall use increases and efficiency drops. While concerning, this pattern creates a healthy ecosystems between commercial software and hardware industries. One creates performance, while the other consumes it, meanwhile the consumer buys more products to maintain ground.

2.7.5 Summary

The design goal of any programming language and its model is to allow the clear and accurate expression of problems in a form that they can be solved computationally[8]. Ideally the programmer should be unburdened by concerns of the computation or hardware model to allow for clarity of expression of the problem domain. Historically it has been difficult to remove hardware concerns from the programming model without harming its computability [43].

Programming models that eschew a hardware model, such as functional and logic programming, are unpopular in the main (2.7.3) [246]. This unpopularity is probably due to the mismatch between ingrained models of computation (from imperative languages) and the forms of expression embodied by these styles of programming. This results in a *perception* of inefficiency or poor expressibility leading to increased development time and lower performance. Inefficiency is a real concern when the programming model eschews hardware concerns. As the transformation from programming language to machine operations becomes more complex and less direct, the harder it is

[8]With the obvious exception of esoteric programming languages such as Brainfuck, Shakespeare or Whitespace.

for the programmer to reason about how their choice of program statements will be executed by the computer. This is important for achieving performance either by design or by optimisation [152].

An important function of the programming model embedded in a given programming language is to guide the programmer into expressing the problem or its solution in a manner which can be computed efficiently by a real computer. That is to say the programming language, paradigm and model should have *mechanical sympathy* with the computer. A key argument of this thesis is that process-oriented programming has a high degree of mechanical sympathy with modern computer systems.

When using the process-oriented programming model, the programmer is free to use concurrency for program structure. This intentional and explicit concurrency aids reasoning by the programmer, and is directly accessible by the runtime system. In other words the programming language and runtime system do not need to extract concurrency in a safe manner from implicit dependencies, as it has direct access to the programmers intentions.

Isolation between processes is an accurate abstraction for distributed and non-uniform memory architectures (2.6). Or rather, even if access to all memory is possible from all parts of the computer system it will have performance implications. Process-oriented programming makes these implications obvious to the programmer as the movement of data between points in the system must occur as explicit communication between processes. This allows the programmer to consider the optimisation step of explicitly reducing the amount of communication, or otherwise restructuring communication. From the runtime system perspective, monitoring access to memory so as to manage implicitly shared resources is computationally expensive to an extent which mitigates its value [148]. However, as the connections between processes in a process-oriented system are explicit these can be used as a source of optimisation (see section 3.4). Where shared memory is available this can be used to efficiently enable communication between processes (see chapter 3).

The current trends in computer hardware are toward distributed memory, many processors and asymmetric multiprocessor architectures (2.6). Modern asymmetric multiprocessor systems, often called *heterogeneous multicore* or *manycore* systems, combine processors with different operating characteristics. The most common example of such systems is a general purpose processor (CPU) coupled with a special purpose graphics processor (GPU). The CPU is efficient when running a sequence of instructions processing a single stream of data and making decisions, the GPU is designed to perform the same sequence of operations on many pieces of data at the same time with few decisions. The mainstream is to address these trends as an optimisation. The programming model is maintained as a single sequential program with a single memory, work is then offloaded to other processors as an explicit optimisation (2.5.16, 2.5.14). Whereas process-oriented programming addresses these by explicit program design (2.4). The gap between single program shared memory models and hardware architectures is only likely to widen. While there is no magic bullet [78], process-oriented programming will maintain mechanical sympathy, as message-passing techniques such as PVM (2.5.23) and MPI (2.5.19) have done in the supercomputer arena for last 20 years [16, 21].

Chapter 3

Scheduling

This chapter describes an efficient scheduler for process-oriented programs written in occam-pi. The occam-pi programming language runtime system is redesigned to enable parallel execution of occam-pi code on modern multi-core computer hardware. The aim of this engineering work is to evaluate the mechanical sympathy of process-oriented software on multi-core computer systems and address the key research question of this thesis: can software written using a process-oriented style of programming be executed efficiently on modern multi-core computer systems using available hardware parallelism?

Content from this chapter has been previously published as [218] and [220].

3.1 occam

As previously noted the occam programming language was designed for developing programs for the Transputer processor (see section 2.5.21). The Transputer embedded a process scheduler, which facilitated the concurrent use of internal and external communication channels. This can in a sense be seen as a form of hardware software co-design, as processor and language are interdependent.

```
-- sequential block
SEQ
  x := x + 1
  y := y + 1
-- parallel block
PAR
  x := x + 1
  y := y + 1
```

Figure 19: Statement level concurrency in occam.

The occam language supports statement level concurrency as shown in figure 19. In the first section a `SEQ` keyword is used indicating that the values of the variables `x` and `y` should be computed and assigned in sequence. In the second section the `PAR` keyword indicates that the same two computations should be executed in parallel. The occam compiler converts these statements into Transputer instructions without making any attempt to optimise granularity. Hence in the example given the `PAR` will cause the processor to create two processes. This fine-grain concurrency means that occam processes should be *lightweight* so as to require a minimal amount of memory and processor time to manage. The Transputer could switch between processes in tens of clock cycles and required only four words of memory per process of management overhead.

The occam compiler disallows memory accesses that would be non-deterministic or unsafe in the presence of concurrency. This means that data which is shared between concurrent processes is read-only, and cannot be used to communicate. Communication between processes is performed through channels. The Transputer allowed any memory word to be used as a synchronous channel, with input and output. Figure 20 shows occam's channel syntax; the concurrent read and write of the channel `c` will synchronise transferring the value 42 via the channel to the variable `x`. Significantly these channels could be mapped to external communication links on the Transputer which could be connected to other processors or devices.

The occam compiler computes the memory usage of programs at compile time. This a priori memory allocation removes the need for runtime memory management, the behaviour of which is non-deterministic. This static analysis guarantees that a program

```
CHAN OF INT c:
INT x:
PAR
  c ! 42 -- write channel
  c ? x  -- read channel
```

Figure 20: Channel syntax in occam.

will never run out of memory during execution. However to enable this the maximum number of processes and the sizes of their associated workspaces must be a compile time constant.

In the 1990s the occam toolchain developed by INMOS and later SGS-THOMSON was adapted to produce executable code for a range of computer systems as part of the occam-for-all project [261, 204]. Two approaches were investigated: modification of the compiler to generate machine code from occam [204] and conversion of occam compiled as Transputer instructions to machine code [261]. Due to difficulties in modifying the monolithic occam compiler the latter of these approaches formed the basis for the Kent Retargetable occam Compiler, *KRoC* [191]. As no other computer processor provides the Transputer's process and channel instructions these are supported by a runtime system called *CCSP* (C Communicating Sequential Processes).

3.2 occam-pi

The occam-pi language significantly extends occam by enabling features which require runtime memory management [48, 257]. Using runtime memory allocation, limitations on the numbers of processes and their respective sizes are removed or eased. Although the compiler must still be able to compute the size of a process's workspace (stack) at compile time[1], objects with dynamic sizes which are only known at runtime can be allocated on the heap.

[1]This does not preclude recursion as a new stack, with a size determinable at compile time, can be allocated from the heap for each level of recursion [48].

Major language additions in occam-pi are *mobile* memory and channels. A mobile piece of data is not copied when it is sent over a channel, rather the reference to it is transferred to the receiver and lost from the sender. If the size of the mobile data is known at compile time it is possible for both the sender and receiver to have memory of the appropriate size allocated such that communication is implemented as the swapping of references. With this scheme the memory requirements of the program can still be computed at compile time just as in standard occam, and `NULL` references are not possible [48]. However if the size of an allocation is only known at runtime, or dynamic allocation is otherwise desired, then mobile data is allocated from the heap.

Mobile channels allow the dynamic reconnection of processes. When one end of a channel is sent over another channel, the sender loses the reference and the receiver gains it. This maintains the invariant that there are only two references to a channel at any given moment, one for reading and one for writing.

There are circumstances where it is desirable to share the end of a channel, for example multiplexing multiple senders. To support this occam-pi adds shared channels to which there can be any number of references. Before a shared channel end can be used for communication it must be explicitly claimed, which establishes the invariant that there is at most one reader and one writer at any given time.

Barriers have also been added in occam-pi and these can also be mobile [255]. Significantly when a mobile barrier is communicated its reference is not lost from the sender. Additionally the receiver is automatically enrolled on the barrier. This maintains synchronisation between sender, receiver and barrier phases.

3.3 Prior Work

Work presented here is not the first multiprocessor implementation of a runtime system for occam. Kevin Vella developed a multiprocessor runtime for symmetric multiprocessor computers [250]. In his work Vella identifies two major issues affecting multiprocessor implementations of lightweight processes such as those required for occam:

- **Shared-memory contention** between processors accessing a *shared run-queue* of processes. A common scheduler design uses a single queue of processes ready to execute. Processes are taken from the front of the queue, run for some period of time, then placed on the back of the queue. Multiprocessor support is added by allowing multiple processors to add and remove processes from this shared queue. This shared queue is very effective in keeping all processors fed with work. However safe modification of the shared queue increases cost due to locking and atomic operations, and if used frequently can have other performance implications such as cache-line contention (2.6).

- **Reduced cache efficiency** when a process is descheduled on one processor and migrates to another then the benefits of any data it has drawn into one processor's cache are reduced. If a process moves rapidly between processors it will gain no benefit from memory caches, significantly hindering its performance (2.6). This concern is partially mitigated by the shared L3 cache on modern multicore processors; however, occam processes have very short dispatch times which exacerbates the problem compared to more coarse-grained threads.

Vella proposed and implemented batching as a solution to both of these concerns. A batch is a group of processes which are executed and migrated together. By grouping processes together the granularity of operations on the shared run-queue is coarsened. Each batch is dispatched a number of times before being returned to the shared run-queue. If the number of times a batch is dispatched is greater than the size of the batch, then all processes in the batch will be dispatched, potentially multiple times. This allows the batch to build up and then benefit from a cache footprint.

Vella's design and implementation has a number of limitations which this work seeks to address.

- The work predates occam-pi and hence does not address dynamic processes, channels or barriers. Additionally the design does not incorporate more recent work by Moores and Barnes on priority [191, 48].

- The implementation only addresses SPARC and DEC Alpha processors. Today's commonly available multiprocessor processors are based on Intel x86 instruction set.
- Batches have a fixed maximum size and the batch of a process is determined at creation. Although a process can move batches through communication Vella does not explore this in his design, focussing on benchmarks which involve constant numbers of processes with minimal communication. This may be because such benchmarks will have the highest potential for linear speed up with respect to Amdahl's law [30] as synchronisation is minimised. Critically if the destination batch is full then a new batch will be created. Eventually batches will become full of *unrelated* processes.
- There is data race in Vella's wait-free communication algorithm with respect to alternation (choice). This can potentially cause memory corruption and other undetermined effects. Details of this race are described along with a new alternation algorithm in section 3.11.
- Process termination in occam requires an implicit barrier between the parent and child process. This is implemented with a spin lock which presents a potential synchronisation bottleneck when multiple large numbers of processes terminate simultaneously. While this bottleneck is unlikely to present performance problems, it highlights Vella's focus on fixed networks or processes. Buoyed by dynamic support in occam-pi, recent usage involves large dynamic process networks [223, 49, 35].

3.4 Design Goals

The overarching goal of this work is to evaluate the mechanical sympathy of process-oriented software with modern multi-core hardware by extracting the maximum performance from unmodified occam-pi program code. To achieve this the runtime system must be as efficient as possible.

3.4.1 Work Stealing

A major limitation of Vella's work is the shared run-queue. Reviewing performance numbers from Vella and Barnes, the growing gap between memory speed and processor speed becomes clear. Vella suggests that for a single processor implementation a context switch takes 580ns or 35 cycles on a 60MHz SPARC, and 200ns or 47 cycles on a 233MHz DEC Alpha [250]. Barnes suggests that on an 800Mhz Pentium 3 a context switch requires 90ns or 72 cycles. Thus with newer hardware processes can be switched more frequently and will be due to reduced execution times; however, the relative cost of switching is increasing. Increased frequency of switching with batches or without batches will increase contention for a shared run-queue. This contention decreases performance linearly with respect to the number of processors contending for the memory, and worse saturates the memory bus with synchronisation traffic which reduces the performance of processors not contending [118]. This is a reflection of Amdahl's law [30].

The point of contention can be removed by using separate run-queues for each processor. The complexity envisaged by Vella is that work must be balanced between these queues. A potential solution is *work stealing* [57, 59]. In a work stealing scheduler processors which become idle (empty their run-queue) steal work from the run-queues of other processors to keep themselves busy.

Research on work stealing has been heavily focused on directed acyclic graphs (DAGs). The Cilk programming language and its associated work stealing scheduler are a prime example of this [58, 57]. Commonly in a DAG a given node or task cannot complete until its dependencies have completed. These dependencies can be located by following the edges of the graph. By enumerating dependencies a scheduling order can be derived: at each node its dependencies are pushed on to a stack. Tasks can then be executed in an appropriate order by removing nodes from the top of the stack. In practice this stack can be built at runtime as a program executes and is always used last-in first-out (LIFO). A LIFO stack can be implemented using wait-free O(1) algorithms [127]. For work-stealing the LIFO is extended to a double-ended queue where work is stolen from the bottom (or end) of the queue which is otherwise used in a LIFO manner. Such a queue can also be constructed in a wait-free manner [126]. These properties make DAGs amenable to wait-free work stealing. In particular there is no need to regularly dispatch all concurrent components in a first-in first-out (FIFO) manner as is required for general purpose process-oriented models such as that used in occam-pi. This allows for the use of a double-ended queue where the cost of work-stealing is taken only on one end of the queue.

A *lock-free* algorithm is one which does not require a lock to be taken before a process or processor enters the algorithm, i.e. the algorithm is not mutually exclusive and can be executed simultaneously by multiple processes or processors [249]. Common definitions of lock-freedom associate it with a guarantee of system-wide progress such that at least one process or processor is guaranteed to make progress executing the algorithm [126]. Where multiple processes or processors may be executing the algorithm simultaneously, but system-wide progress cannot be guaranteed then the algorithm is referred to as *non-blocking* [126]. A *wait-free* algorithm is a lock-free algorithm where any process or processor can complete the algorithm in a finite number of steps, regardless of the execution speeds of any other processes or processors operations, i.e. no operation can starve another [127]. Thus wait-free algorithms offer the strongest guarantees of progress and non-blocking algorithms offer no guarantees. These guarantees refer to the worst-case performance of the algorithms. Performance in the common case may often be improved by relaxing guarantees [40].

Process-oriented programs and more specifically occam programs are not acyclic graphs. A process network is in fact a directed multigraph, with the additional property that processes can execute indefinitely. This contrasts with most DAG systems such as Cilk where a task will complete in finite time should its dependent tasks complete. For occam programs a first-in first-out (FIFO) queue represents the ideal scheduling case. Assuming that processes yield to the scheduler in finite time, this guarantees that all processes will get executed at some point.

Vella considered wait-free algorithms for the shared run-queue in his design, but decided against them due to increased complexity and the lack of CAS or LL/SC (2.6.1) on SPARC v8. The performance of non-blocking algorithms is robust (does not degrade) in systems with multiple processors [182]. Non-blocking FIFO queue algorithms are lock-free or non-blocking, but not wait-free [249, 181]. Although wait-free queues have recently been developed, non-blocking variants have better performance due to lower numbers of atomic operations [153].

While general purpose wait-free algorithms may not be efficient [40], there are certain constraints that can be used when designing an algorithm for a work stealing scheduler. The most beneficial aspect of work-stealing is bias. The scheduler stealing work is otherwise idle and hence can afford to spend more time acquiring work, whereas a scheduler operating on available local work should be suffer minimal interference. Hence algorithms should be biased toward the scheduler operating locally, the common case in a loaded system. Additionally unlike a fully shared run-queue, only one scheduler will ever add work to any given queue. This gives three distinct operations for the run-queue: *local enqueue*, *local dequeue* and *remote dequeue*. Local operations occur frequently and must be cheap, remote operations are infrequent and can be more expensive. All operations should have defined upper bounds on numbers of operations to complete.

3.4.2 Working Set Batches

Batches provide a effective mechanism for improving the cache utilisation of light-weight processes such as those found in occam-pi [250, 88]. With a work stealing scheduler a run-queue of batches continues to have the benefit of reducing potentially costly enqueue and dequeue operations. A difference is that batches will only migrate when the system is unbalanced and has capacity rather as a side effect of passing through the run-queue as is the case with a shared run-queue. This means finer-grain smaller batches can be considered without negatively impacting performance. Vella's design gives little consideration to the make up of batches. Therefore this is an obvious area for refinement.

Within a process network a given process communicates with only a subset of processes as defined by the channels to which it is connected. Its immediate peers represent the processes with which it engages in communication and hence synchronisation. If these synchronisations can occur on the same processor then cache ping-pong effects can be avoided. Hence it is desirable for frequently communicating processes to be scheduled together on the same processor.

This observation holds true for dynamic networks such as the very large process networks used in complex modelling simulations exemplified by those produced by the CoSMoS project [3, 35, 222]. Typically agent processes move by reconnecting channels between themselves and different space processes. In any given simulation step a number of agent processes will be communicating with a single space process. This represents a snapshot working set.

Static analysis would not be useful for these dynamic networks, as only runtime measures can determine the connectivity of a given process. Tracking which channels a process has access to is possible; however, the associated data structures would be inefficient to maintain and use. Therefore monitoring which channels a given process communicates on seems the most appropriate option for determining working sets. Communication between processes should draw them in to the same batch. Note that

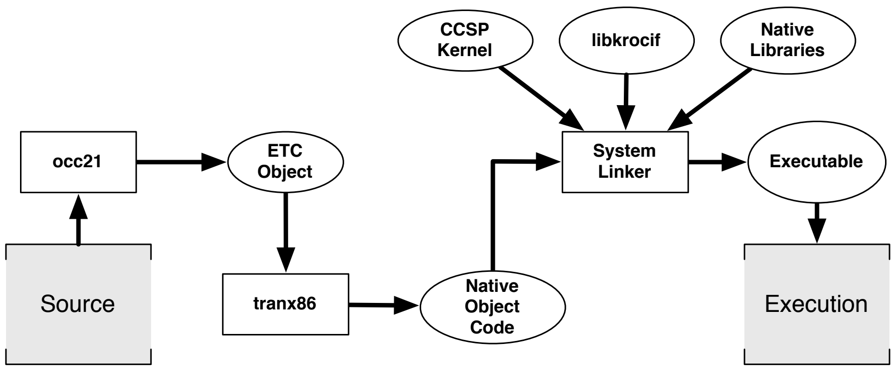


Figure 21: KRoC toolchain structure.

to accomplish this there should be no maximum batch size. Counter to the above, processes that do not communicate should eventually end up in separate batches. These two pressures should be embodied in the scheduler behaviour so as to produce batches which contain distinct working sets and adapt to changing process network topologies.

3.5 KRoC

This works builds on the KRoC toolchain as of 2007 [204, 261, 191, 48]. The toolchain structure can be seen in figure 21. KRoC is composed following components:

- occ21, a modified version of the INMOS/SGS-THOMSON occam compiler. This takes occam or occam-pi as input and produces Extended Transputer Code (*ETC*) as output [205]. ETC is an intermediate representation holding Transputer instructions in a binary form, but retaining symbolic links and other information necessary or useful for producing an assembled executable.

- tranx86, which translates ETC into Intel x86 assembly language [47]. A conversion between the 3-place stack machine architecture of the Transputer and the register architecture of x86 is performed. This does not produce machine code; this task is delegated to a system compiler and linker such as GCC [5].

- CCSP, the runtime kernel which supports channel communication and task scheduling [191].
- libkrocif, a library providing bootstrap and support functionality for occam-pi programs, for example terminal input and screen output.

There are some subtle details of the toolchain which must be addressed in order to add multiprocessor support. The majority of these are associated with occam-pi features and the means by which they have been added.

In adding features to occam to create occam-pi Barnes took the approach of minimising additions to the runtime scheduler and virtual instruction set [48]. Essentially only three groups of new virtual instructions were added:

- Heap allocation instructions to support mobiles such as `MALLOC` (memory allocate) and `MRELEASE` (memory release).
- Mobile communications such as `MIN` (mobile input) and `MOUT` (mobile output).
- Semaphore operations to support shared channels, `SEMINIT` (semaphore initialise), `SEMCLAIM` (semaphore claim) and `SEMRELEASE` (semaphore release).
- Barrier operations to support static and mobile barriers such as `BARINIT`, `BARSYNC`.

Semaphore operations are converted by tranx86 into machine instructions. Details of the semaphore's waiting queue are managed outside of the runtime scheduler, with more generic `RUNP` (run process) and `OCCSCHEDULER` (invoke scheduler) instructions used to start and stop processes on the queue. A similar approach is applied to barrier operations. These conversions abstract details from the runtime scheduler and weaken the knowledge the scheduler has of a program's intentions. Additionally it is not possible to change details of process descriptors without modifying tranx86 or occ21 both of which manipulate these directly.

While the runtime scheduler is aware of the movement of mobile types by channels through `MIN` and `MOUT` instructions it is unaware of the contents of the mobile. This is significant for mobile channel ends which must be reference counted in order to prevent memory leaks. The location of the reference count is determined by code in occ21 and operations to modify the count inlined. These operations will not be safe (unless made atomic) in a multiprocessor environment. Similarly mobile barriers must enroll the receiver atomically when communicated via a channel.

Essentially the runtime application program interface (API) between generated code and the runtime kernel (CCSP) is weakened as knowledge of runtime structures and memory management is embedded in both occ21 and tranx86. This prevents modification of runtime details without modification of all parts of the toolchain. Thus to support work on the runtime scheduler the API was strengthened to remove runtime and scheduling details from occ21 and tranx86. A new strong API is presented in appendix A. Modifications were made to occ21 so that all integration with the runtime is done via this API and tranx86 converts all associated operations to calls to the runtime scheduler. Additionally an object model is defined in appendix B. This gives the runtime scheduler and occ21 a common structure for mobile data. Memory management details are abstracted from occ21, with occ21 simply requesting particular types of mobile memory with associated parameters. The runtime is now responsible for operations such as reference counting.

A further detail to be addressed is identifying the active scheduler. The CCSP runtime uses static memory to store its run-queues. This must be changed to allow for a scheduler structure for each processor or OS thread. Having done this it is necessary to identify the current thread in which the scheduler has been invoked. This can be done using *thread local storage*; however, there is a cost associated with resolving the address of variables held in thread local storage which is incurred each time such a variable is used [98]. While this cost involves a read from memory and a few address calculations, it still desirable to avoid this overhead in the critical path. Instead the runtime was modified to overload the call table, placing the scheduler structure at a fixed offset from it.

During compilation tranx86 converts instructions which invoke the runtime scheduler into calls through a lookup table. While this call table is not required as kernel calls can be linked directly [47], it is used to reduce x86 instruction length. The address of the call table is held in a machine register ('ESI' on x86) and kernel calls perform an indirect jump to offsets from this register. By optimising the order of calls in the table, frequent operations (e.g. channel input or output) can be placed as low offsets. In turn these indirect jump instructions are smaller than to linked memory addresses. This improve pipelining and instruction cache efficiency on Intel Pentium 3 and Pentium 4 processors. This low-level optimisation is unnecessary on more recent processors; however, by replicating the call table for each processor and placing the associated scheduler data structure at a fixed offset from this it is possible to resolve the location of the current thread's runtime scheduler using a single arithmetic instruction.

3.6 Algorithms and Conventions

All algorithms presented in this chapter assume a total store order memory model (2.6.1) [254, 199]. Unless otherwise specified all variables are machine words (32 bits on Intel x86). Low order bit positions for example 0 and 1 refer to bits representing small magnitude integers such as 2^0 and 2^1. Conversely high order bit position represent high magnitude integers such as 2^{31}. The position of the bits within the word bytes with respect to endianness is not relevant.

Several common operations are used in algorithms:

- load, read – `value` ← `*address`

  Load a *value* from a memory *address*.

- load, read – `value` ← `address[field]`

  Load a *value* from a memory *address* offset by *field* words.

- set, store, write – `*address` ← `value`

  Store a *value* to a memory *address*.

- set, store, write – `address[field]` $\leftarrow$ `value`

  Store a *value* to a memory *address* offset by *field* words.

- atomically swap – `Swap(address, new)`

  Atomically store the *new* value at *address* and return the *old* value. This can be implemented by `LOCK; XCHG` on Intel x86.

- compare and swap – `CAS(address, old, new)`

  Atomically store *new* value at *address* if the contents of *address* is equal to the *old* value. The return is *true* if the exchange happens otherwise *false*. This can be implemented by `LOCK; CMPXCHG` on Intel x86.

- atomically decrement and test – `DecrementAndTest(address)`

  Atomically decrement the value at *address* and detect if the value is zero. This can be implemented by `LOCK; SUBL` followed by `SETZ` on Intel x86.

- atomically test and set – `TestAndSet(address, index)`

  Atomically set the bit at *index* of *address*, returning the old value. This can be implemented by `LOCK; BTSL` on Intel x86.

- atomically test and clear – `TestAndClear(address, index)`

  Atomically clear the bit at *index* of *address*, returning the old value. This can be implemented by `LOCK; BTRL` on Intel x86.

- read barrier – `ReadBarrier()`

  Prevent read pipelining past this point. This is an *LFENCE* on Intel x86.

- memory barrier – `MemoryBarrier()`

  Flush all memory pipelines. This is a *MFENCE* on Intel x86.

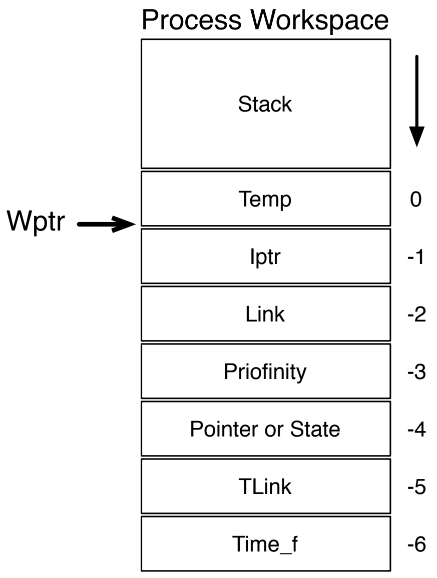


Figure 22: Layout of process workspace showing fields of process descriptor and their associated word offsets.

3.7 Processes

Each process has a *process descriptor* used to store state when descheduled or performing certain kernel calls. The descriptor can be allocated statically on the process stack, or when state does not need to persist across kernel calls it may be allocated dynamically at the point of call. For occam-pi the process descriptor is stored in the process workspace and the compiler guarantees that the workspace is large enough to hold the descriptor. The process is hence referenced by its workspace pointer, `Wptr`. The process stack is accessible at positive offsets from the `Wptr` and the process descriptor fields at negative offsets. Figure 22 shows the layout of the workspace and process descriptor.

The process descriptor fields are used as follows:

- `Temp` a temporary area used to communicate some information between kernel calls and the process. This is the only process descriptor field visible to the process. In practice it is only used for alternation to indicate which channel is ready.
- `Iptr` is the address of the instruction to resume at when the process is rescheduled

or the kernel call finishes.

- `Link` is a pointer to another process descriptor when the process is on a queue, or `NotProcess_p` if it is the end of the queue.
- `Priofinity` holds the priority and affinity mask of the process. The affinity mask describes which logical processors a process can execute on. This is useful for binding processes to parts of the system. An empty mask means the process can run on any processor; this is equal to a full mask, but the full mask has a special behaviour as described in 3.8.7.
- `Pointer` or `State` holds the data pointer during channel communication or the alternation state during alternation.
- `TLink` is a pointer to a timer queue node if the process is waiting or alternating on a time offset.
- `Time_f` is the time offset being waited if the process is waiting or alternating on a time offset.

The value of these fields are preserved by the process between kernel calls; for example the process may allocate or deallocate stack overwriting data in these words. Therefore the state of these fields (with the exception of `Iptr`) is undefined at the beginning of any kernel call, but can be relied upon as long as the process is held in the scheduler. The exception to this is alternation where the compiler is disallowed from allocating or deallocating stack between beginning an alternation and finishing it, as the process descriptor maintains state between kernel calls. This means that operations that occur between `ALT` and `ALTEND` can rely on the value of these fields.

Assuming a process must have at least one word of data; these fields mean the minimum size a process workspace is 8 words (32 bytes on a 32-bit machine). In practice the minimum memory allocation supported by the runtime is 64 bytes which is aligned so as to occupy exactly one cache-line. This helps to avoid false sharing of cache-lines, although the compiler is free to allocate arbitrary sized pieces of process workspace to subprocesses. Ultimately this minimal memory overhead makes the creation of very large numbers of processes practical.

3.7.1 Scheduling

For each physical processor, core or hardware thread in the host system a scheduler instance, a *logical processor*, is started. The logical processor contains a number of run-queues, each of which is a linked list of *batches*. Batches are in turn linked lists of process descriptors, linked using the `Link` field. The structure of the logical processor is shown in Figure 23. It is divided into two sections, internal fields:

- `dispatches` holds the remaining process dispatches for the current batch, also known as the *dispatch count*.
- `priofinity` is the current priority and affinity of the logical processor; this is used to fill in the `Priofinity` field in the process descriptor.
- `curb` holds the present batch.
- `timeout` is the time at which the next timer queue node becomes ready.
- `timer queue fptr` is a pointer to the front of the timer queue.
- `timer queue bptr` is a pointer to the back of the timer queue.
- `runqueue state` is a bitmap in which each bit represents whether a run-queue contains batches.
- `runqueue[P]` are the run-queue structures, one for each priority level; there are P priority levels.
- `free` a LIFO stack of free scheduler data structure allocations. All scheduler data structures (batches, timer queue nodes, etc) fit within a single cache-line (on 32-bit systems). By maintaining a pool of available memory, scheduler structures can be

rapidly allocated with only three memory operations. As the pool is a LIFO, hot cache-lines will be reused first, reducing cache traffic.

- `laundry` is a LIFO stack of scheduler structures which are being used by another logical processor. Periodically these are scanned and available structures placed on the `free` stack.

External fields:

- `sync` a bitmap used to signal external events to the logical processor. This is updated with atomic test-and-set or test-and-clear instructions. It is checked periodically by the scheduler.
- `bmail` is a FIFO queue of batches, a batch mailbox. This is used to send batches to the logical processor.
- `pmail` is a FIFO queue of processes, a process mailbox. This is used to send processes to the logical processor.
- `mwstate` is the migration window state bitmap, similar to the `runqueue state` bitmap it indicates which migration windows contain batches.
- `mwindow[P]` are migration windows which mirror (some) batches held in the `runqueue`. Again there is one for each run-queue, P in total (one for each priority level).

The logical processor executes a scheduler loop which has three components:

1. Check `sync` and handle events. These events will either require mailboxes to be emptied or the timer queue to be checked.
2. If the current batch has finished, retire the batch and select a new one. This may involve work stealing if the logical processor has no batches.
3. Dispatch the next process from the `curb` (current batch).

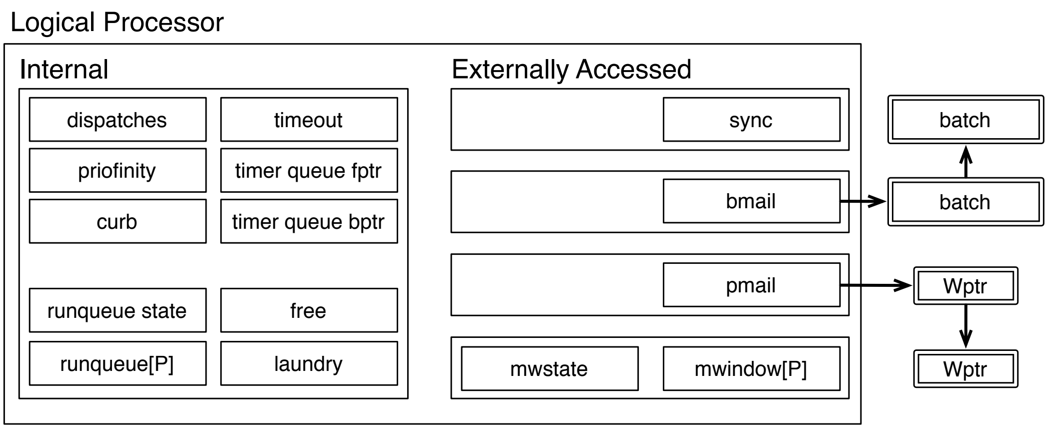


Figure 23: A logical processor instance schedules batches of processes on each physical processor. The structure is divided into internal and external parts.

The scheduler executes each batch by copying its details to the `curb` or *active queue*. The `dispatches` count is calculated based on the number of processes in the batch (multiplied by a constant) and bounded by the *batch dispatch limit* (constant). The count is decremented each time a process is taken from the `curb` and executed. When the count reaches zero, and `curb` is not empty, a new batch is allocated and the contents of `curb` copied into it. This batch is then added to the end of the appropriate run-queue.

3.7.2 Batches

Batches are stored in the data structure shown in figure 24. The fields are used as follows:

- `fptr` points to the first process in the batch.
- `bptr` points to the last process in the batch.
- `size` is the number of processes in the batch.
- `next` points to the next batch when this batch is on queue or stack.
- `state` records the batch's offset when it is placed in a migration window and whether it is dirty (its pointer is shared) when a batch is migrated.

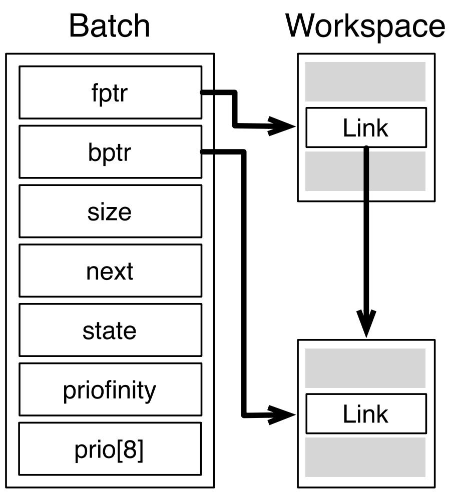


Figure 24: Batch data structure layout and fields.

- `priofinity` is the priority and affinity information of processes in this batch.
- `prio[8]` is used for priority caching in barrier algorithms (see 3.14).

The scheduler attempts to group processes into the same batch when they communicate or synchronise with each other. By forming batches in this way, processes which communicate frequently are scheduled on the same processor, reducing interprocessor traffic. This is an improvement to Vella's techniques. Variable size batches are formed and split automatically using runtime heuristics.

Following a context switch, if the dispatch count is not zero, then the next process on the active queue is dispatched. Otherwise the scheduler restarts with a new batch. Context switches occur under two conditions. Most commonly, the current process blocks on a communication or synchronisation primitive and is descheduled. Alternatively, a process may cooperatively yield to the scheduler, in which case it is placed at the end of the active queue. With the exception noted in section 3.7.3, processes rescheduled (unblocked) by the currently executing process, for example by the completion of communication, are also placed on the end of the active queue. It is this action which draws related processes into the same batch.

3.7.3 Batch Size

If processes are always drawn into a batch during creation and communication, then one batch will eventually grow to encompass all processes in the system. This will prevent batching from having caching benefits as the working set will contain all active processes. Therefore a mechanism is required to prevent batches growing too large and to separate processes which lose association.

We observe that in independent subgraphs of a process-oriented program network there will be points when only one process in the subgraph is active. This process reschedules other processes in the subgraph which may then in turn become the only active process. For example, imagine a process which produces data from user input and initiates computation in other processes. Eventually the computation completes and the other processes cause the display graphics to be updated. Consider such a process network alongside a pipeline in which execution (the control flow) follows each piece of data through the pipeline from process to process. These subgraphs can execute in parallel as long as they do not communicate with each other. If they communicate then one subgraph will wait for the other, i.e. only one process will be active.

Based on the above observation we state that if while executing a batch there is a point at which only one process is active then that batch is probably optimal, i.e. it contains one independent subgraph or thread of control flow. Conversely batches which do not meet this condition during execution should be *split*. Batches are split by placing the head process of the active queue in one batch, and the remainder in another. This is a unit-time operation, and so can be carried out frequently. Repeated execution and split cycles quickly reduce large and unrelated batches to small related process subgraphs. Erroneous splits will quickly reform based on the other scheduling rules.

Additional mechanisms to control batch size can be introduced by modifying the dispatch count in response to specific events. Process creation is one example. During process creation the new process is placed on the end of the active queue. Process creation does not cause a context switch; however, the runtime kernel decrements and tests the dispatch count. This prevents the batch size exceeding the dispatch count. Furthermore, if the dispatch count reaches zero and the aforementioned conditions for batch splitting are met, then a process creating many new processes will be split into a separate batch from the newly created processes. The newly created batch is then free to migrate (see section 3.8). Thus a process spawning a large number of children may continue to execute while its children begin execution on other logical processors in the system.

3.8 Run-queues and Process Migration

This section describes how logical processors interact as part of a multiprocessor system. As logical processors have separate run-queues, work is distributed between logical processors via migration. Processes are free to migrate between logical processors, except where restricted by an explicit affinity setting. Migration occurs in two circumstances:

1. A process which blocks during communication or synchronisation and is descheduled on one logical processor can be rescheduled by a process executing on a different logical processor. Unless prohibited by an affinity setting, the rescheduled process continues execution on the rescheduling logical processor.
2. A logical processor which runs out of batches to execute may *steal* batches from other logical processors [59, 88].

The first case occurs as part of the communication (3.9) and synchronisation (3.13) algorithms. The second case is the mechanism by which work is spread across the system. It is further underpinned by the observation that independent long-running subgraphs of processes will tend to be split into separate batches, which can be stolen by idle logical processors.

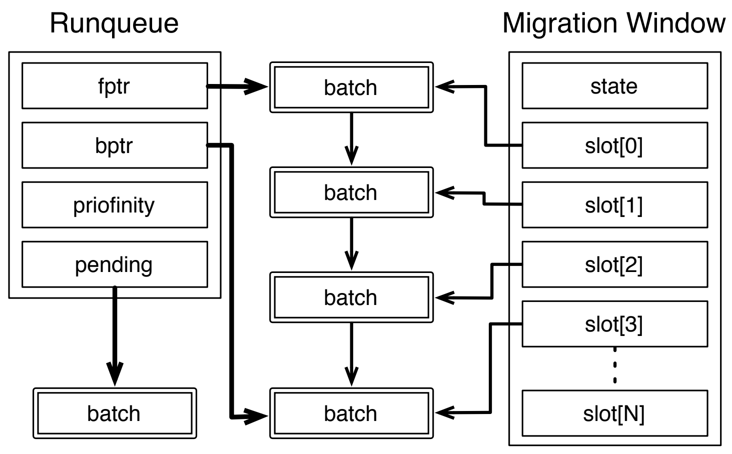


Figure 25: A fixed-size migration window array allows one logical processor to "steal" batches from another.

The run-queues of each logical processor are private and cannot be accessed by other scheduler instances. To allow batch migration, a fixed-size *migration window* provides other logical processors with access to the end of each run-queue. The fixed size of the window allows it to be manipulated using wait-free algorithms [129, 127]. These provide freedom from starvation and bounded completion when contention arises, improving scalability over locks.

Figure 25 shows the relationship of the migration window and run-queue. The `fptr` and `bptr` provide a linked-list of batches on the queue. The `pending` field always points to a batch; this batch is used to gather processes rescheduled during execution which do not belong in the current batch (`curb`) in order to maintain priority and affinity invariants. Batches in the migration window must not be modified (to avoid data races), hence the `pending` batch is held separately. The `priofinity` reflects the priority and affinity of the pending batch.

The `state` field of the migration window contains a bitmap of active slots, and a counter indicating which slot is the head. When manipulating the migration window the logical processor stores new batches at the head offset before incrementing it and updating the bitmap with a normal write. If a new batch replaces an old one then that batch is no longer in the window and cannot migrate through work stealing. A remote scheduler stealing work uses the bitmap to search for batches to attempt to steal. On successfully stealing a batch the remote scheduler updates the bitmap using an atomic *test-and-clear* operation. This design provides a bias allowing the local scheduler to update the state field without atomic operations. If the local scheduler overwrites a bitmap update from a remote scheduler this only reduces the efficiency of the bitmap by marking a slot active when it is in fact empty, an inconsistency which can be detected when the slot is manipulated.

The bitmap limits the size of the migration window size to 27 slots on a 32-bit processor (5 bits used for head offset). For simplicity a 15 slot migration window is used in the implementation presented here as large amounts of work stealing are not expected. If a larger migration window is required then the bitmap could be dropped and a linear scanning approach used for remote dequeue without affecting the correctness of the algorithms presented here. The size of the window limits the number of batches that can be stolen from a given logical processor between run-queue iterations. If the window is too small then it is possible that batches will not be available for idle logical processors to steal and not all processors in the system will be able to find work (for a short period).

Algorithm 1: Run-queue local enqueue algorithm (wait-free).

```
1  link batch into the run-queue linked list
2  if batch has affinity then
       // enqueue is complete
3      return
4  end
5  load the window state word
6  generate a new offset by incrementing the last offset (handling roll-over)
7  read window slot at the offset
8  if result is null then
       // no one else will write to slot
9      simply store batch into slot at offset
10 else
11     atomically swap batch pointer into slot
12     if result is not null then
           // batch has been knocked out of window, it is clean
13         clear state field of knocked out batch
14     end
15 end
```

3.8.1 Local Enqueue

The algorithm 1 is used to place a batch onto the run-queue of a logical processor and make it visible in the migration window.

Internal operations on the window will be more common than external operations, hence the presented algorithms are optimised for the uncontended case rather than the contended case. The effect of this optimisation is that the final step of the algorithm can produce corruption of the window state word. In the event of corruption the window will appear to external logical processors to contain more batches than it does; however, this does not affect correct operation of the external dequeue algorithm (only its operating efficiency). The result is an algorithm with a deterministic number of steps and at most one (potentially) expensive atomic operation. On completing the enqueue algorithm, the logical processor updates its `runqueue state` and `mwstate` bitmaps to indicate that the associated run-queue has batches.

3.8.2 Local Dequeue

To dequeue a batch from its run-queue, a logical processor uses the algorithm 2.

While the dequeue algorithm may fail and have to restart, it is bounded by the number of batches enqueued on the logical processor and the size of the migration window. In the worst case, every batch may have been stolen and the scheduler must scan every batch to discover this. However at most *migration window size* batches can be stolen giving a deterministic upper bound to execution. This local scanning does not create direct contention with other logical processors, but processors may still contend for underlying system resources such as the memory bus.

Algorithm 2: Run-queue local dequeue algorithm (wait-free).

```
1  remove the head batch from the run-queue
2  if no batch stored at window offset then
       // batch is not in window; dequeue completes
3      return batch
4  end
5  compare and swap null with the migration window slot
6  if compare and swap failed then
       // batch has been stolen by an external scheduler
7      place batch on laundry queue for later cleanup
       // dequeue has failed
8      if run-queue is not empty then
9          restart
10     else
11         return null
12     end
13 else
       // dequeue complete
14     update migration window bitmap
15     return batch
16 end
```

In the event that the run-queue is empty the logical processor should take the pending batch if it contains processes. The pending batch is replaced with an empty one.

3.8.3 Remote Dequeue

When one logical processor attempts to steal work from the migration window of another, it does so using the algorithm 3. This algorithm requires only two atomic operations in the optimal case.

Having migrated a batch the logical processor copies the contents to a new local batch data structure and marks the original batch as clean by atomically clearing the *dirty* bit in its `state` field. The pointer to the batch is then discarded. The originating logical processor will later collect the original batch structure and reuse it. This allows each logical processor to maintain its own pool of batch structures, and minimises cache ownership contention (inverting the scheme creates higher cache traffic, see 3.8.4).

Algorithm 3: Run queue remote dequeue and batch theft algorithm (wait-free).

```
load the window state word (creating a local copy)
rotate the active bitmap by the last offset
while local copy bitmap is not empty do
    scan the bitmap to select an entry to steal
    atomically swap null into window slot
    if result is not null then
        // got a batch
        atomically clear the relevant bit in the window state word
        return batch
    else
        // batch has already been taken
        clear associated bitmap bit in the local copy
    end
end
// migration fails
return null
```

3.8.4 Laundry

As noted in the algorithms 2 and 3 a batch is placed on the `laundry` queue when it is found to have been stolen. It is not removed from the `laundry` until the *dirty* bit is found to be cleared. This can be done by scanning the `laundry` queue periodically checking the `state` fields of batches. This behaviour handles the brief period when an remote scheduler has the pointer to a batch and is still copying it. Once the dirty bit is clear the remote scheduler has finished with the batch and it can be used again locally.

Alternatively the remote scheduler could assume ownership of batches it steals, and the local scheduler would discard references to batches that are found to be stolen. However the local scheduler would still need to signal to the remote scheduler that it has removed the batch from its queues, which may not happen for a considerably longer period of time than it takes for a batch to be copied.

Additionally the laundry behaviour presented minimises cache-line ping-pong effects in the common case. Consider the sequence of events:

1. local scheduler enqueues the batch
2. remote scheduler steals and copies the batch (`fptr`, `bptr`, `size` and `priofinity`)
3. remote scheduler clears the dirty bit
4. local scheduler reads the window offset from the `state` field
5. local scheduler accesses the slot and discovers the batch stolen
6. local scheduler tests the dirty bit in the `state` field and finds the batch clean

This represents the ideal and common case (as batch copying is fast). The cache-line containing the batch only needs to move twice between steps 1 and 2, and 3 and 4. This is optimal.

3.8.5 Work Stealing

When a scheduler has no work in any of its local logical processor run-queues it must steal work. This is done by scanning all other logical processors for work, reading the migration window state `mwstate` of these logical processors. The logical processor with the highest priority available work is selected and the scheduler attempts to steal a batch from its highest priority run-queue with work. This continues until a batch is stolen or all available logical processors are exhausted of work. If no work is available the scheduler busy waits before trying again. After a number of attempts the processor sleeps.

Whenever a scheduler selects a new batch from the logical processor it checks whether it has excess work available for migration by testing the `mwstate` bitmap. If work is available the scheduler tests a global bitmap of sleeping logical processors. The first sleeping logical processor is then woken up so it can perform work stealing. By starting one sleeping logical processor at a time, thrashing from multiple processors attempting simultaneous work stealing is avoided.

3.8.6 Priority

This implementation preserves the semantics and behaviour of process priority as implemented by Barnes [48]. A separate run-queue is used for each priority level. Queues are serviced in priority order, with lower priority queues not dispatched until higher priority queues are exhausted. If a higher priority run-queue than the current becomes available, e.g. by resuming a suspended high priority process, the present batch is saved and the scheduler switches to the higher priority run-queue. Batches only ever contain processes of the same priority.

As detailed in section 3.8.5, the work stealing algorithm respects priority by attempting to migrate high priority work first. If work stealing is not triggered then it is possible for logical processors to be running at different priority levels. However any interaction between high priority and low priority processes will cause the low priority process to be pre-empted.

3.8.7 Affinity

As previously mentioned process affinity is supported. This can be used to bind a process to a set of logical processors. This is stored together with the priority to form the `Priofinity` field of a process. Batches only ever contain processes with the same affinity bitmap. Batches of affine processes (processes with processor affinity) are not placed in the migration window, meaning a scheduler cannot accidentally steal work it cannot execute.

Whenever a scheduler reschedules a process which has an affinity bitmap which does not include its itself the process is sent directly to one of the logical processors in the affinity bitmap. This is done via the `pmail` queue of that logical processor and may cause it to be woken if it is sleeping.

An empty affinity bitmap, the default, means a process can execute anywhere. This is equivalent to a full affinity bitmap except that a full bitmap disables work stealing of that process. Additionally as processes only ever occupy batches with other processes with the same affinity bitmap, the affinity bitmap can be used to subtly influence the batching mechanism. In practice this behaviour has not been used.

3.8.8 Blocking System Calls

Barnes added language and runtime support for blocking system calls [48]. These allow an occam process to invoke a piece of external code which runs in a separate operating system thread. Through this mechanism the external code runs concurrent to occam processes executing the runtime scheduler thread.

In Barnes' model the language invokes the runtime kernel to dispatch a blocking system call. A thread pool of *dispatch threads* is used to reduce or eliminate the overhead of allocating a thread for each call as threads can be reused. A complication is that on completion of the system call the dispatch thread must return the process to the CCSP run-queue: this is a race hazard.

In the modified CCSP run-time presented here, the race hazard with process return is trivially removed. On completion of a blocking system call the dispatch thread sends the process batch to a logical processor via its `bmail` (batch mailbox). The batch mailbox is used rather than the process mailbox because the process workspace cannot be used to store a process descriptor as the compiler does not allocate sufficient memory to store the `Priofinity` word. Thus a batch is used as the `Priofinity` is stored in the batch structure not the process descriptor.

Affinity is respected by only mailing the batch back to a suitable logical processor. The thread pool is also divided to allow a single shared pool for requests with no affinity mask and a per-processor pool available for affine requests. Each affine request pool has at most one dispatch thread, and requests for this thread are queued. This means that if a process sets an affinity mask its blocking system calls will only ever executed on a single thread, and requests for that thread serialised. This is useful for external libraries or functions which are not re-entrant, thread-safe or contains other race hazards.

3.8.9 Mailboxes

Enqueue and dequeue from process and batch mailboxes (`pmail` and `bmail`) is wait-free with algorithm 4 and algorithm 5. Each mailbox has a front pointer `fptr` and a back pointer `bptr`. The algorithms shown refer to a `node` which has a `next` pointer. For batches the `next` pointer is the `next` pointer in the batch. For processes the `Link` field is used as the `next` pointer.

Both algorithms are wait-free and rely on the fact that only one logical processor ever dequeues from the front of the queue. Only nodes which have the `next` pointer set or are pointed to by both `fptr` and `bptr` can be dequeued. This allows detection of only partially queued nodes which cannot be dequeued as they will be updated (when the `next` pointer is written). Thus the dequeue algorithm can fail if a node is only partially enqueued.

This dequeue failure is not a problem as the presence of mail is not signalled until the enqueue completes. Likewise the mail signal flags in the `sync` word of the logical processor are cleared (by atomic swap) prior to attempting a dequeue. Thus a partially complete enqueue at the time of dequeue will always raise a new signal when it completes.

Algorithm 4: Mailbox enqueue algorithm (wait-free).

```
node[next] ← null
node′ ← Swap(bptr, node)
if node′ = null then
    *fptr ← node
else
    node′[next] ← node
end
```

Algorithm 5: Mailbox dequeue algorithm (wait-free).

```
node ← *fptr
if node ≠ null then
    bptr′ ← *bptr
    if bptr′ = node then
        if CAS(bptr, node, null) then
            CAS(fptr, node, null)
            return node
        end
        ReadBarrier()
    end
    node′ ← node[next]
    if node′ ≠ null then
        *fptr ← node′
        return node
    end
end
return null
```

3.9 Communication

Interprocess communication is central to process-oriented programming, for sharing state and synchronising computation. The efficiency of communication therefore directly affects the performance of process-oriented designs.

The runtime kernel provides a single basic communication primitive for processes to exchange data: point-to-point synchronised channels. Synchronised channels require no buffers and data is copied or moved (depending on the mode of operation) directly between the source and destination processes. Buffered channels can be constructed efficiently by placing buffer processes between communicating processes. Transactions involving many parties sharing a channel are implemented by associating the channel with a mutual exclusion lock (3.13).

Operations for channel input and output take a source or destination buffer and a size in bytes to copy. Alternatively the source and destination may be a reference to a mobile type (appendix B) allocated through the runtime kernel, in which case the reference is moved between the processes together with ownership of the object. If the mobile type is data, then the reference is moved and deleted from the source, maintaining the invariant that occam-pi programs are free from aliases to mutable data. When a channel end is communicated, the type is checked and if it is not shared then it is treated as mobile data. If it is shared then the reference is copied and the reference count atomically updated. On communication of a barrier type, the receiver is automatically enrolled on the barrier (and its reference count maintained).

A channel is represented by a single machine word. The word stores a pointer to the process descriptor (3.7) of the process waiting to communicate on the channel. The process descriptor is guaranteed to be word-aligned, allowing use of the low order bits in the channel word for communicating other information. For the algorithm which follows only the *alternation bit* is relevant. It indicates whether the process descriptor stored in the channel is *blocked* on this channel or *waiting* on a number of channels and events (3.11).

Algorithm 6: Channel communication algorithm (wait-free).

```
 1 read the channel word
 2 if word is null or has alternation bit set then
 3     store the process state in the process descriptor
 4     store the destination or source buffer pointer in the process descriptor
 5     atomically swap the process descriptor with the channel word
 6     if result is null then
 7         context switch occurs, a new process is selected as in section 3.7
 8         return
 9     else if result has alternation bit set then
10         trigger the alternation event as in section 3.11.8
11         return
12     else
           // original read was stale, continue
13     end
14 end
   // channel word is not null, a process is blocked on the channel
15 load the destination or source buffer pointer from the blocked process descriptor
16 copy data or move references and transfer ownership
17 reset the channel word to null
18 reschedule the blocked process
19 return
```

Basic channel communication, regardless of direction, is performed using algorithm 6. Using this algorithm the second process to reach the channel completes the synchronisation and thus the communication. This results in, typically, only one of the two processes performing an atomic operation.

3.10 Timers

The occam-pi language supports timers which can be read and waited on. In order to support these timers, CCSP provides a machine word size clock which ticks every microsecond. Where possible this is implemented using the CPU instruction counter. Figure 26 shows an example of using a timer in occam.

When a process waits on a timer (waiting for a specific point in time to pass) a timer

```
TIMER t:
INT now:
SEQ
  t ? now                    -- read time
  t ? AFTER (now PLUS 1000)  -- wait 1ms
```

Figure 26: Timer syntax in occam.

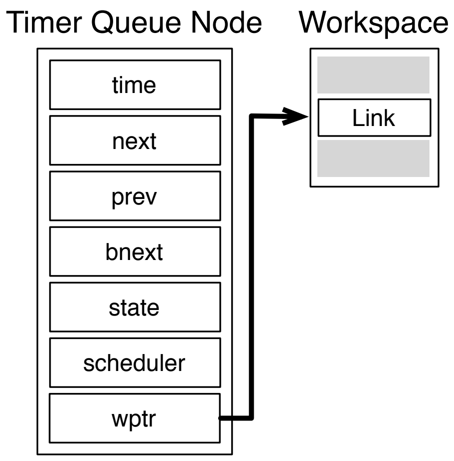


Figure 27: Timer queue node structure.

queue node is allocated for it. The structure of the timer queue node can be seen in figure 27. This node is stored on an ordered double linked timer queue on the current logical processor. Each logical processor has its own timer queue (`timer queue fptr` and `timer queue bptr`). The `time` field holds the time at which the node expires. The node fits in a batch structure and can be treated as a batch (for laundry purposes). To support this the `bnext` field is provided (for linking on the laundry stack) along with the `state` field to hold the dirty bit. The `scheduler` field points to the logical processor where the node is queued; this allows optimisation when an alternation with a timer node completes on the same logical processor as its timer queue node is allocated on. The `wptr` field points to the process descriptor (process workspace), it is used to provide interlock as the `wptr` is atomically swapped when the node expires.

The expiry time of the next timer queue node is cached in the logical processor's `timeout` field. The scheduler regularly tests whether `timeout` field of the logical processor has passed. By default this occurs between batches or when a timer interrupt occurs (an operating system signal). If the timeout has expired then the timer queue is walked and processes whose timeouts have passed are rescheduled. If only the timer queue is populated and no further work is available then the scheduler uses operating system calls to sleep an appropriate amount of time.

3.10.1 Timer Expiration

When the timer queue entry expires, the scheduler for that timer queue executes algorith 7. This algorithm must interface with possible timer alternations, as described in section 3.11. If a process completes a timer queue alternation on another logical processor, then the `wptr` pointer is stolen by that logical processor. The timer queue node is also placed on the `laundry` stack of the other logical processor. It is then the responsibility of the logical processor which created the node to discover this, mark the node clean and forget the reference to the node. To facilitate this the `state` field of the node is marked `dirty` when it is allocated.

3.11 Alternation

For many purposes, blocking channel communication is sufficient; however, processes often need to multiplex between a number of channels and other events. CCSP supports *choice* (2.3.3) over a number of channels and timer events. This is called *alternation* and supports the occam `ALT` language construct. Figure 28 shows an example alternation in occam.

Alternation allows a process to wait for one of a set of channels to become ready. When an element of the waited set becomes ready, the process is rescheduled and can make a choice as to which channel to communicate with. This is similar to the POSIX `select` system call.

Algorithm 7: Timer expiry algorithm (wait-free).

```
1  unlink the timer queue node from the timer queue
2  if wptr is null then
       // process stolen by alternation
3      set batch state to clean
4  else if wptr has alternation bit set then
       // active alternation
5      atomically swap null with wptr field
6      if result is not null then
7          execute event trigger algorithm on result (algorithm 12)
8      end
9      set batch state to clean
10 else
       // no alternation
11     store the current time into the process descriptor Time_f
12     reschedule the process
13     place timer queue node on free list
14 end
```

```
CHAN OF INT chan.0, chan.1:
TIMER time:
INT x:
ALT
  chan.0 ? x
    -- x receives value from chan.0
  chan.1 ? x
    -- x receives value from chan.1
  time ? AFTER (now PLUS 1000)
    -- 1ms passed and no channel became ready
```

Figure 28: Alternation syntax in occam.

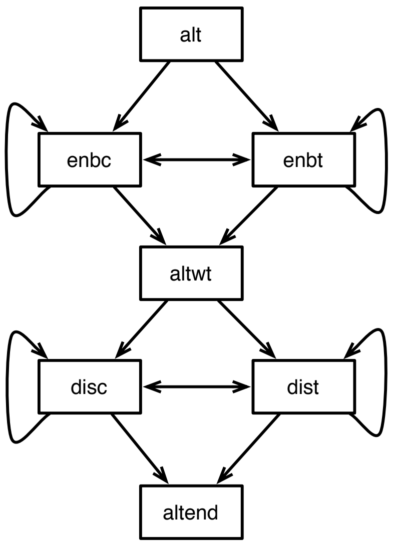


Figure 29: Alternation instruction order.

This section presents algorithms designed for one process waiting on a set of exclusively input or output channels, while other processes sharing those channels commit. This constraint is enforced by the present version of the occam-pi language which only allows alternation on input. When two or more processes wait on overlapping sets of channels a race hazard exists (2.3.3). More general synchronisation algorithms are part of ongoing research.

Alternation is a multiple stage process. Figure 29 illustrates the instruction sequence and figure 30 the associated state transitions. First the process executes an `ALT` instruction which sets up the process descriptor's initial state. The process is now in the *enabling* state. Then a number of `ENBC` (enable channel) and `ENBT` (enable timer) instructions setup the events the process is waiting for. At any point in this sequence one of the events can already be ready or become ready causing the process state to transition to *ready*. Having enabled all events, the `ALTWT` (alternation wait) instruction is executed. If no events are ready then the process is in the *waiting* state and is descheduled. When the `ALTWT` completes the process will be in the *ready* state as at least one event is ready. The process disables channels and timers with `DISC` and `DIST` instructions. Finally the

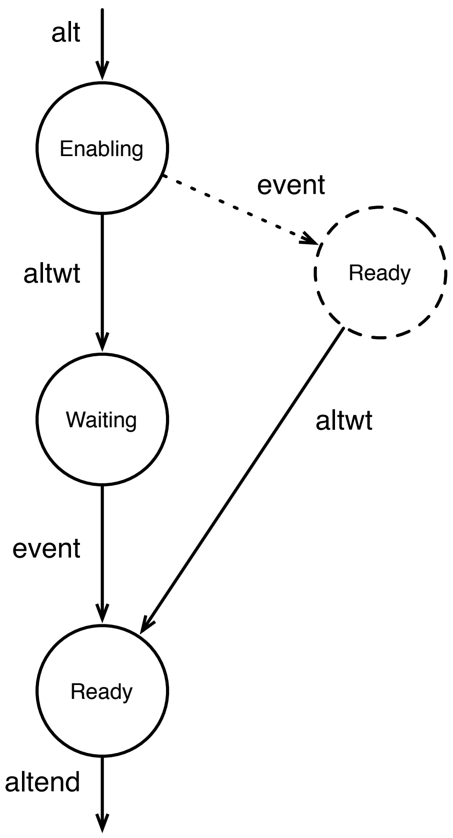


Figure 30: Alternation state transitions.

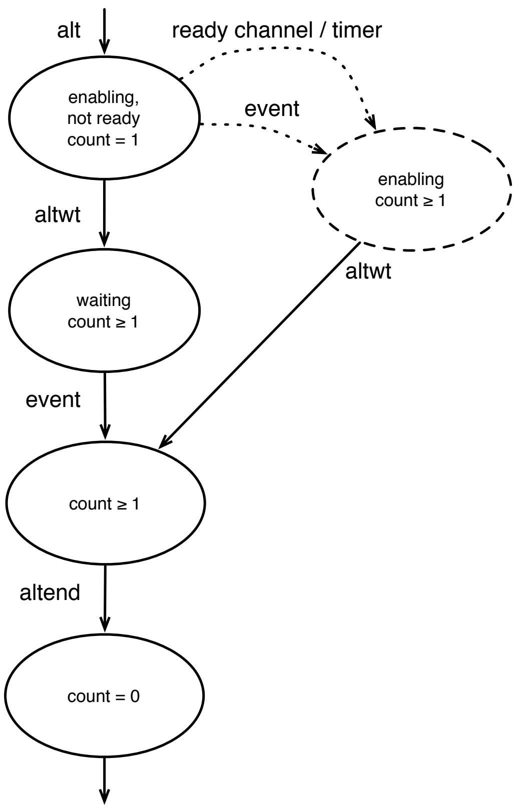


Figure 31: Revised alternation state transitions.

`ALTEND` instruction cleans up remaining state and causes the process to branch to code for handling the event which became ready. The order in which channels are enabled or disabled can be used to implement priority in selecting events [52].

Historically the state of the process has been stored in the `State` field of the process descriptor. During channel communication the party completing the communication reads the `State` field of the process descriptor of the blocked process. If the value is one of the three states *enabling* (`1`), *waiting* (`2`) or *ready* (`3`) the blocked process is alternating, otherwise it is a regular communication and the `State` field is a `Pointer` field.

In Vella's multiprocessor implementation the same state model is used [250]. During a channel communication:

1. the alternating process descriptor is retrieved by atomically swapping the communicating process descriptor into the channel word;

2. the alternation state read is read from the alternating process descriptor and then potentially atomically updated.

These are two distinct actions which can be interrupted by operating system pre-emption. If this happens then the process descriptor pointer retrieved during 1 may no longer point to a process descriptor at step 2. During the pre-emption any sequence of events can occur on another scheduler including the completion of the alternation after which any sequence of operations may follow. To avoid this a way of knowing about pointers to the alternating process descriptor held in pre-emption is required.

An important observation is that the number of possible pointers to the alternating process descriptor is bounded by the number of channels (and timers) enabled. The algorithms presented here use this to reference count the alternating process descriptor. The process descriptor must be maintained as long as the reference count is greater than zero. Maintenance of the process descriptor is achieved by preventing the alternation from completing until all references have returned or been deleted.

The reference count is stored in the `State` field. This prevents reads of the process descriptor `State` field being sufficient to distinguish an alternating process. This is because a `Pointer` may point to any byte in system memory. Low values such as 1, 2 and 3 address the first word of memory which will almost certainly be invalid on most systems (to maintain C `NULL` pointers as invalid). The same is not true of larger values. Instead an alternating process is signalled to the communication algorithm by an *alternation bit* in the channel word. This also allows branching of the communication algorithm to the alternation specific code based on the value of the first read of the channel word.

To further simplify parts of the algorithm the alternation state is changed to be composed of distinct condition bits rather than a numeric state.

- `enabling` bit signifies that the alternation is in the state before `ALTWT`.

- `not ready` bit signifies that no events are ready before waiting.
- `waiting` bit signifies the process is descheduled and waiting for events

These bits are stored in the `State` field and unlike a numeric state can be updated independently with *test and set* atomic operations if required. They allow identification of the state transition path where a concurrent event causes an alternation to become ready while enabling.

The revised state transitions are illustrated in figure 31. Initially the state is `enabling`, `not ready` and has a reference count of 1. Any action by the process or external event (e.g. other process communicating) that causes a channel or timer to become ready clears the `not ready` bit. The `ALTWT` (alternation wait) signals the end of the enabling phase and clears the `enabling` bit. If the `not ready` bit is set when `ALTWT` occurs then `waiting` is set and the process is descheduled. The `waiting` bit indicates the process must be rescheduled if an event occurs. In either case when a channel or timer is ready the disabling state is reached, this is signified by the absence of other bits. Disabling operations reduce the count by collecting references, the `ALTEND` (alternation end) collects any references which have been stolen. Having completed an `ALTEND` the reference count will reach zero.

3.11.1 Initialisation

Initialisation sets up the process descriptor for subsequent alternation operations. As the process descriptor has not yet been shared, protection from races is not required. The `State` field of the process descriptor is initialised. The alternation state consists of:

- *flags* indicating what stage of alternation the process is in. The initial flags are `enabling` and `not ready`.
- A *reference count* which tracks the number of pointers to the process descriptor, initially one. When a logical processor triggers an event which is part of an alternation it takes one of these references. The alternation only completes when

all references have been counted back through the disable algorithm or via event triggers.

If required (in a timer alternation) the `TLink` and `Time_f` fields are also initialised.

3.11.2 Channel Enabling

Each channel a process alternates over is enabled using algorithm 8.

Algorithm 8: Channel enabling algorithm (wait-free).

```
1  read the channel word
2  if word is not null then
3      atomically clear the not ready flag of the alternation state
       // enable operation completes indicating the channel is ready
4      return ready
5  else
6      atomically swap a pointer to the process descriptor with the alternation bit set
       into the channel word
7      if result is not null then
           // value from initial read was stale
8          write the result back to the channel
9          return ready
10     else
11         atomically increment the reference count in the State field
12         return not ready
13     end
14 end
```

3.11.3 Timer Enabling

Each timer that is enabled updates the `Time_f` field of the process descriptor. This is done such that the `Time_f` represents the earliest point in time, as this will be the first to trigger. If the timer value has already passed then the `not ready` flag of the alternation state is cleared.

3.11.4 Waiting for Events

Once the process has enabled all the events it makes the `ALTWT` (alternation wait) kernel call. An atomic compare and swap is used to clear the `enabling` and `not ready` flags, and set the `waiting` flag. If the compare-and-swap succeeds then the process is descheduled and a context switch occurs. Failure indicates that an event has become ready, in which case the `enabling` flag is atomically cleared and execution of the process continues.

If any timers were enabled during the alternation the `Time_f` field is rechecked for expiry during the `ALTWT`. If the time has not passed then a timer queue node for the process is created. A reference to the timer queue node is stored in the `TLink` field.

3.11.5 Channel Disabling

Having been woken up, the process disables channels using algorithm 9.

Algorithm 9: Channel disabling algorithm (wait-free).

```
1  read the channel word
2  if word does not contain a pointer to the process descriptor of the alternating process
   then
       // channel is ready
3      return ready
4  else
5      compare and swap null into the channel
6      if compare and swap fails then
           // channel just became ready
7          return ready
8      else
           // channel is not ready; reclaim reference
9          atomically decrement the reference count in the State field
10         return not ready
11     end
12 end
```

3.11.6 Timer Disabling

The process executing the alternation disables timer events using algorithm 10. This algorithm interacts with details of the timer structure (3.10). Timer events return ready if the time queue pointer is set to *time set* and the timer timeout is after the time specified when disabling the event. An optimisation to the algorithm removes the atomic swap when the timer queue node is part of the timer queue of the runtime scheduler on which the process is currently executing, and additionally removes the timer queue node from the timer queue. When modifying timer queue nodes which are on a scheduler other than the current they cannot safely be removed from their associated timer queues, so are reaped later by signalling the scheduler that it has dead timer queue nodes.

Algorithm 10: Timer disabling algorithm (wait-free).

```
1  read TLink field of process descriptor to get timer queue node
2  if TLink is null then
       // timer queue node was never allocated
3      return not ready
4  end
5  atomically swap null with the wptr in the timer queue node
6  if result is not null then
       // timer event has not fired
7      atomically decrement the reference count in the State field
8      save "time not set" constant to the TLink field
9      place the timer queue node on the laundry stack
10     return not ready
11 else
       // timer event has fired
12     copy time field of the node into the Time_f field of the process descriptor
13     save "time set" constant to the TLink field
14     place the timer queue node on the laundry stack
15     return ready
16 end
```

3.11.7 Finalisation

Having disabled all channels and timers, the alternation is finalised using algorithm 11. This completes the alternation and after this operation communication with any ready channels may take place.

Algorithm 11: Alternation finalisation algorithm (wait-free).

```
1  read the reference count from State field of process descriptor
2  if count is not one then
3      save the process state as if to context switch
4      atomically decrement and test the reference count
5      if count is not zero then
6          deschedule alternating process and context switch
7          return
8      end
9  end
   // alternation is finalised
10 return
```

3.11.8 Event Trigger Algorithm

Logical processors execute the *event trigger algorithm* (algorithm 12) to signal an alternating process that one of its waited events has become ready. This is the algorithm referenced in channel communication, algorithm 6. If a logical processor in the event trigger algorithm holds the final reference then it is responsible for rescheduling the alternating process.

3.12 Extended Rendezvous

Barnes added extended rendezvous to occam-pi [48], which is similar to Ada's extended rendezvous (2.5.2). This allows a receiver (or sender) to block the other party in the communication after synchronisation, creating a temporary mutual exclusion such that the initiator of the rendezvous can reason that the other party is not running.

Algorithm 12: Event trigger algorithm (wait-free, bounded on number of channels).

```
1 repeat
2     read the State field of the process descriptor to trigger
3     generate a new state with the not ready and waiting flags cleared
4     compare and swap new state into State field
5 until compare and swap succeeds;
6 if original state has waiting flag set or the new reference count is zero then
7     add the process descriptor onto a run-queue, rescheduling the alternation
      process
8 end
```

```
CHAN OF INT c:
INT x:
SEQ
  c ?? x
    SEQ
      -- x has been received, but sender is still blocked
      ... use x ...
  -- sender is been released
```

Figure 32: Extended rendezvous syntax in occam.

Figure 32 shows an example of extended rendezvous in occam-pi. Figure 33 shows execution of processes performing an extended rendezvous.

The CCSP runtime supports an extended rendezvous as a single channel alternation, and a modified communication algorithm for the initiator. The initiator makes a XABLE (rendezvous enable) call, which is the same as ALT (alternation initialise), ENBC (enable channel) followed by ALTWT (alternation wait). The actual state of the alternation does not need to be tracked beyond this. If the channel is ready then the process continues execution, otherwise the process will be rescheduled by the channel communication algorithm (algorithm 6). Having become active again the process performs communication using a modified communication algorithm which does not reschedule the other process or remove its process descriptor from the channel word. To release the other process the initiator makes an XEND (rendezvous end) call which loads the process descriptor from the channel word, clears the channel and reschedules the other process.

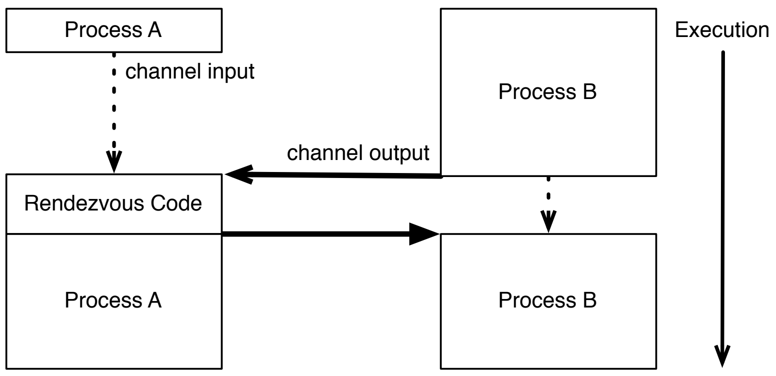


Figure 33: Execution of processes performing extended rendezvous. Process A performs an extended input and waits for Process B to arrive. On arrival, Process B is blocked and Process A is rescheduled. On completion of the rendezvous Process B is also rescheduled.

3.13 Mutual Exclusion

Section 3.9 describes communication channels capable of synchronous point-to-point exchanges involving pairs of processes; however, there are designs which require multiple communication peers to use the same channel. This is particularly useful for implementing the deadlock-free client-server design pattern [259], in which a number of clients communicate with a single server over channels.

To support this functionality mutual exclusion locks can be associated with the channel directions. This allows ordered multi-access (used by multiple processes) channels to be constructed. The lock *claim* and *release* algorithms are lock-free[2] and prevent starvation using FIFO queuing. Importantly, the occam-pi compiler enforces claim and release semantics on these locks, so that a programmer cannot forget to release the channel lock.

The mutual exclusion lock is represented by a queue with a front pointer (`fptr`) and back pointer (`bptr`). In a sense the mutual exclusion lock is an enhanced version of the mailboxes used for process and batch communication (3.8.9), but where the present lock holder dequeues processes.

[2]Claim operations can block the process, but are lock-free from the perspective of the logical processor.

The lock is taken using algorithm 13 and released using algorithm 15. These algorithms both use algorithm 14 for dequeuing processes from the semaphore queue. Due to looping behaviour these algorithms are *lock-free* not *wait-free*: a faster processor can detect that a slower processor is stalled on the queue, abort and retry. In the case where an enqueue interferes with a dequeue, the enqueue will recover the lock and reschedule a process from the queue. This means that although one scheduler can fail to dequeue, another will always succeed.

The lock itself is represented by the lowest order bit in the back pointer, if this bit is set the lock is available, otherwise it is held by a process. Storing this bit in the back pointer means release of the lock can interfere with operations to add processes to the queue. This is intentional as release of a lock should cause another scheduler attempting to queue a process to try the lock again, rather than enqueue the process.

Algorithm 13: Mutual exclusion lock claim algorithm (lock-free).

```
1  p ← *bptr
2  if p = 1 then
       // lock available
3      if CAS(bptr, 1, 0) then
           // lock acquired
4          continue process execution
5      end
       // reload back pointer
6      p ← *bptr
7  end
   // lock not available; attempt to queue process
8  while not CAS(bptr, p, Wptr) do
9      p ← *bptr
10 end
   // p holds old back pointer; update front pointer
11 if p = 0 or 1 then
12     *fptr ← Wptr
13 else
14     p [Link] ← Wptr
15 end
16 if lowest bit of p is set then
       // got lock via CAS
17     attempt dequeue and reschedule algorithm 14
18 else if TestAndClear(bptr, 0) then
       // got lock by atomic operation
19     ReadBarrier()
20     attempt dequeue and reschedule algorithm 14
21 end
```

Algorithm 14: Mutual exclusion lock dequeue algorithm (lock-free).

```
fptr' ← *fptr
if fptr' $\neq$ 0 then
    bptr' ← *bptr
    lptr ← fptr' [Link]
    if bptr' = fptr' then
        *fptr ← 0
        if CAS(bptr, fptr', 0) then
            // dequeue successful
            return fptr'
        end
        // enqueue interleaved
        *fptr ← fptr'
        restart
    else if lptr $\neq$ 0 then
        // dequeue successful
        *fptr ← lptr
        return fptr'
    else
        // create pointer to fptr' [Link]
        p ← fptr' [Link]
    end
else
    // create pointer to fptr
    p ← fptr
end
// partial enqueue is blocking dequeue; release lock
TestAndSet(bptr, 0)
MemoryBarrier()
p ← *p
if p $\neq$ 0 then
    if TestAndClear(bptr, 0) then
        // got lock back; restart
        ReadBarrier()
        restart
    end
end
// dequeue failed
return 0
```

Algorithm 15: Mutual exclusion lock release algorithm (lock-free).

```
1 atomic p ← *bptr
2 if p = 0 then
3     if CAS(bptr, 0, 1) then
          // lock released
4         return
5     end
6 end
7 attempt dequeue and reschedule algorithm 14
```

3.13.1 Operation and Correctness

The lock is claimed by setting the lock bit (in `bptr`) from 1 to 0. Algorithm 13 initially tests for the simplest case: lock is available and queue is empty (line 2). In this case the algorithm attempts to claim the lock (line 3). If this does not succeed the process is added to the back of the queue by atomically setting the `bptr`. If the queue was empty then the front pointer `fptr` is set (line 12), else the link pointer is updated process descriptor which was previously the back of queue (line 13). Once this has completed, the old value of `bptr` is tested. If the lock bit is set then the lock has been claimed incidentally and algorithm 14 is invoked. Otherwise a final attempt to claim the lock is made using a `TestAndClear` operation (line 18).

On completing algorithm 13 one of three possible states is reached:

1. The current process has claimed the lock and is still running.

2. The current process has been added to the end of the lock queue, and the lock is still held by another process. In which case the scheduler will dispatch another process.

3. The current process has been added to the end of the lock queue, and the lock is available. In which case algorithm 14 is invoked.

Algorithm 14 removes the first element of the queue. This algorithm is only executed by one logical processor at a time (the one holding the lock); however, while execution of the algorithm is not interleaved, removing processes from the lock queue requires the detection of interference from enqueue operations. First the front pointer of the queue (`fptr`) is tested; this will only be set if at least one process has been fully enqueued (line 2). The back pointer of the queue (`bptr`) is loaded (line 3), along with the link pointer of the front queue node (line 4). If the queue has a single element then the front and back pointers will be the same (line 5).

If the queue has only a single element then `fptr` is set to `null` (line 6). Only this algorithm will be updating `fptr`; algorithm 13 only operates on the `fptr` when the queue is empty. The `bptr` is set using a compare-and-swap operation to detect interference with an enqueue from algorithm 13. If this operation fails then `fptr` must be restored and the algorithm restarted. As the interfering operation is not affected then its progress is guaranteed.

If the queue has multiple elements then the link pointer must be valid (line 12). It will be `null` if algorithm 13 has not completed an enqueue, i.e. a *partial enqueue*. If the link pointer is valid then `fptr` is updated and dequeue succeeds (line 13 and 14).

If the front pointer is `null` (line 2) or the link pointer is `null` (line 12), then the lock is released (line 21) and operations serialised with a memory barrier (line 22). The respective pointer is then reloaded (line 23). If the value is still `null` then the stalled enqueue has not completed and dequeue fails; when the stalled enqueue completes it will claim the lock and initiate a dequeue. Otherwise the stalled enqueue has finished and the lock is reclaimed (line 25) and the algorithm restarted; unless the lock has been taken by another logical processor.

Algorithm 15 releases the lock by setting the lock only in the case that the queue is empty, i.e. `bptr` is `null`. Otherwise it invokes algorithm 14 to dequeue the next process. In this way the lock is passed from the active process to the next waiting process preserving FIFO ordering of lock operations.

3.14 Barriers

The occam-pi and CCSP runtime support barrier synchronisations. Processes can *enroll*, *resign* and *synchronise* on barriers. Processes synchronising on a barrier are blocked until all other processes enrolled on the barrier also synchronise. Barriers may also be communicated by mobile reference over channels, atomically enrolling the receiver as part of the communication; this permits semantics such as those described by Welch and Barnes [255].

Barriers of this type are useful when implementing agent simulations. Each agent is enrolled on a common barrier and synchronises on it to maintain time-step with the other agents in the simulation. With many thousands of agents synchronising, the performance of barrier operations is critical. It is also important to minimise the time between barrier completion and returning to the state where all enrolled processes are scheduled for execution across available logical processors.

The root barrier structure consists of a state word, which describes the current state of the barrier including the number of processes yet to synchronise, a bitmap of active scheduler pointers, followed by an array of pointers to scheduler specific data. To reduce the number of atomic operations and contention between schedulers on the barrier each logical processor maintains private queues of blocked processes in batches within the barrier.

The structure of the barrier can be seen in figure 34. The only shared part of the barrier is the `state` word. The highest order bit is used to indicate the barrier is *syncing*. The two bits below this represent the *tag* for separating barrier synchronisations. The remaining bits are the *count*.

Each logical processor allocates a *barrier head* which fits in a batch structure (allowing the reuse of memory allocations). A pointer to this is stored in a dedicated slot in the barrier structure. This pointer can be safely stored and read without concern for other schedulers. The `head bitmap` is atomically updated once when the barrier head is stored.

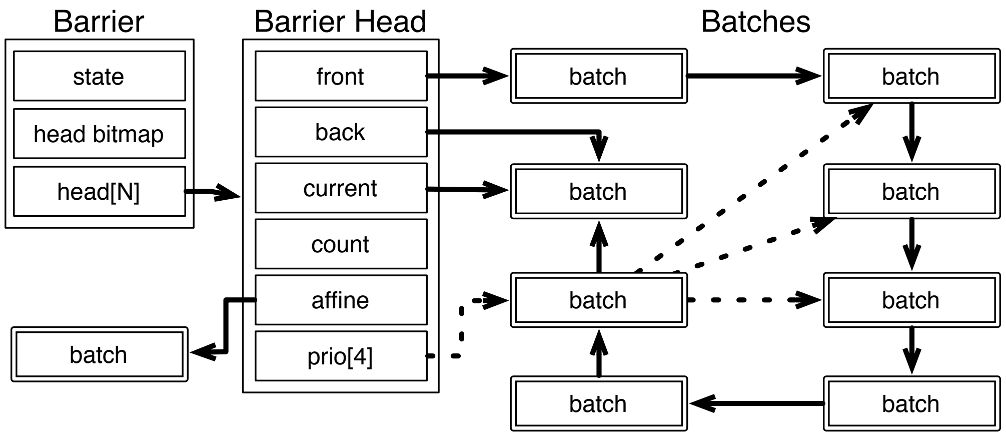


Figure 34: Internal structure of barriers.

The *tag* in the barrier `state` word identifies operations occurring before and after a barrier completion. Barrier head pointers are stamped with the current tag value (in the low-order bits) when they are attached to the barrier. The tag is incremented on barrier completion. This allows the completion algorithm (algorithm 20) to detect barrier head pointers which require completion (old tag), or have already been completed by another logical processor (new tag). In principle only two tag states are required; however, using four states (two bits) allows for debugging.

3.14.1 Enrollment

Enrolling processes on the barrier is trivial. The state word is simply atomically incremented by the number of processes being enrolled. This is *wait-free*. Storing flags in the high-order bits of the state word simplifies this operation as they need not be considered by the atomic increment.

3.14.2 Resignation

Resignation is more complex than enrollment as the barrier count may be reduced to zero and the barrier completed. Algorithm 16 is used to resign processes.

If the count would be reduced to zero by the resignation (line 2) then the algorithm initiates barrier completion. If interference is encountered (line 8 or 14) then the algorithm is restarted. Restarting the algorithm creates a loop; however, interference from resignation and synchronisation operations is bounded by the number processes enrolled on the barrier. Therefore in the absence of enrollment the resignation operation will complete in a finite interval (when all synchronisation and resignation operations complete) [3]. If enrollment causes interference then the enrollment operation is guaranteed to make progress ensuring system-wide progress.

Algorithm 16: Barrier resignation algorithm (lock-free).

```
1  read the barrier state
2  if count is equal to number of processes resigning then
3      generate a new barrier tag (the next in sequence, wrapping)
4      new state = 1 | tag | syncing
5      compare and swap old state with new state
6      if success then
           // barrier completed
7          call barrier completion algorithm (algorithm 20)
8      else
           // interference
9          restart
10     end
11 else
12     new state = state − number of resigning processes
13     compare and swap old state with new state
14     if failed then
           // interference
15         restart
16     end
17 end
```

[3]Assuming that for any two competing compare-and-swap operations one makes progress.

3.14.3 Synchronisation

When a process synchronises on a barrier it is placed on the barrier queue before the barrier count is updated and barrier completion detected. The major complexity involved with synchronisation of a barrier is that additional processes may be enrolled at any point by another scheduler. In particular one scheduler may enroll processes on the barrier which in the process of being completed by another scheduler and these then begin synchronising on the barrier. The algorithms outlined here provide interlock to detect and work safely with these scenarios, at the cost of added complexity.

When a process synchronises on a barrier the algorithm 17 is used to access the barrier head (see figure 34). Once the barrier head has been located the process being queued on the barrier is stored into a batch linked to it. The size of the linked batches is maintained at approximately one-eighth the size of the barrier. This is to prevent barrier completion flooding the migration window. Processes of different priorities and affinities are separated into different batches. A fixed size hash table is used to accelerate lookup of the present batch for any given priority; this uses the `prio` slots of the barrier head and batch structures.

Having queued the synchronising process in a batch on the barrier head, the scheduler executes algorithm 18 to decrement the barrier count and potentially complete the barrier. This is similar to resignation. Once the barrier count has been updated the completion algorithm (algorithm 20) is invoked if the syncing flag was set as part of the state change. The syncing flag essentially locks barrier against further completion, preventing concurrent execution of the completion algorithm. Algorithm 19 gives an overview. Combined these algorithms are *lock-free*.

3.14.4 Completion

To complete the barrier the completing scheduler must gather all the queued processes and their batches then locally schedule them or distribute them to suitable schedulers if affine. The barrier state (including the count of enrolled processes) is rebuilt while

Algorithm 17: Barrier head access algorithm (wait-free).

```
1  read the state
2  read the head pointer slot indexed by the logical processor number
3  if syncing flag set in state word, the head pointer is not null and the low order bits of the
   head pointer do not match the tag then
4      atomically swap null into the head pointer
5      if result is not null then
6          reschedule batches in barrier head as in the barrier complete algorithm
           (algorithm 20)
7      end
8  end
9  if head pointer is null or became null then
10     allocate the barrier head
11     place the tag (from the state) into the low order bits of the pointer
12     store the pointer into the head pointer slot
13     atomically set the relevant bit in the head bitmap
14 end
15 mask out the low order bits of the head pointer
16 return head pointer
```

Algorithm 18: Barrier synchronisation count update algorithm (lock-free).

```
   // use the previously read barrier state, if it has changed
      (unlikely) the compare and swap operation which follows will
      detect this
1  repeat
2      if looping then
3          reload state
4      end
5      if count is one then
           // this it the last process
6          generate a new barrier tag (the next in sequence, wrapping)
7          new state = 1 | tag | syncing
8      else
9          new state = state − 1
10     end
11     compare and swap old state with new state
12 until compare and swap succeeds;
```

Algorithm 19: Barrier synchronisation algorithm (lock-free).

```
1 get barrier head (algorithm 17)
2 enqueue process on to barrier head
3 increment the count in the barrier head
4 update barrier count (algorithm 18)
5 if syncing flag set then
6     complete barrier (algorithm 20)
7 end
```

rescheduling the barrier processes. These steps are performed by lines 2 through 20 in algorithm 20. Once this is complete all processes blocked on the barrier have been rescheduled. The state word can then be unlocked by removing the syncing flag and one from the count (line 30 through 39). Line 23 detects if the barrier has been completed again during the execution of the algorithm, if so the algorithm must restart. Critically this algorithm is only executed by one logical processor, effectively creating a producer consumer relationship with logical processor which enqueue processes during completion.

Algorithm 20 is *lock-free* as termination is not guaranteed for a slow processor, but a slow processor will not block faster ones. It is possible that while completing the barrier all rescheduled processes will be stolen by other schedulers, execute and be queued on the barrier again before the completing scheduler removes the syncing flag (resetting the barrier). In this case the algorithm restarts as if the barrier needs completing again (which it does). This can repeat indefinitely. However in practical terms it is highly unlikely given the respective speed of work-stealing, process execution and the barrier completion algorithm that this will happen unless the completing process is pre-empted by the operating system. It is also unlikely that the scheduler will be repeatedly pre-empted such that it can never finish the completion algorithm before work is stolen and completes. Even if this does happen, processes are still being executed by other logical processors and thus system-wide progress is guaranteed.

Algorithm 20: Barrier completion algorithm (lock-free).

```
1  atomically swap zero with the head bitmap
2  foreach head slot marked active in head bitmap do
3      read head slot
4      if tag in head pointer is not equal to previous tag (cached) then
           // another scheduler has already taken and processed the old
              barrier head
5          skip this slot
6      end
7      compare and swap null into the head slot
8      if failure then
           // another scheduler has already taken and processed the old
              barrier head
9          skip this slot
10     end
11     atomically increment the state by the count in the barrier head
12     foreach batch in the barrier head do
13         if batch is affine then
14             mail batch to an appropriate logical processor
15         else
16             add the batch to a local run-queue
17         end
18     end
19     release the barrier head
20 end
21 repeat
22     load the barrier state
23     if count is one, tag is unchanged, syncing flag is set and head bitmap is not zero then
24         generate a new barrier tag (the next in sequence, wrapping)
25         generate a new state from old state with with new tag
26         compare and swap old state with new state
27         if success then
               // the barrier completed again before algorithm completed
28             restart algorithm
29         end
30     else
31         generate a new state from old state
32         count is decremented by one and syncing flag cleared
33         compare and swap old state with new state
34         if success then
               // barrier completion
35             return
36         end
37     end
38 until barrier completion;
```

3.14.5 Correctness

Interaction on the barrier structure occurs via the barrier state word, head bitmap and barrier head pointers.

The state word is only updated using compare-and-swap or atomic increment operations. This prevents any loss of state from concurrent updates. Execution of the completion algorithm is controlled by the last synchronising process setting the syncing flag. This is done via a compare-and-swap operation which will fail only if more processes are enrolled; enrollment is the only possible concurrent operation at that the point of barrier completion. The barrier tag is updated in the same operation as the syncing flag maintaining its consistency. Barrier completion is mutually exclusive; during barrier completion only enrollment, resignation and synchronisation operations will take place.

The head bitmap is set atomically by logical processors performing synchronisation. It is cleared atomically only by the logical processor performing completion. The same is true of the barrier head pointers. As such each bitmap bit and head pointer is only set by one logical processor and cleared by another. This produces a simple producer consumer relationship. Each logical processor produces its head bitmap bit and head pointer which are then consumed by the logical processor executing the completion algorithm.

During completion the head bitmap is cleared before any processes are rescheduled, hence there is no concurrent access. The head pointers have the barrier tag embedded and attempts to rewrite the pointer (in the synchronisation algorithm) can detect when it has not been cleared (by the completion algorithm). Overwrites from the synchronisation algorithm undertake the action of the completion algorithm (rescheduling processes). This allows the synchronisation algorithm to proceed ahead of the completion algorithm. The completion algorithm only acts upon on head pointers with the appropriate tag. In this manner each barrier head data structure is only used by one logical processor in each barrier phase and does not require further synchronisation.

3.15 Evaluation

This section presents benchmark results evaluating and comparing the performance of the modified occam-pi runtime. The source codes for these benchmarks are publicly available [2].

3.15.1 Test Setup

All benchmarks were performed on an eight core Intel Xeon workstation composed of two E5520 quad-core processors running at 2.26GHz. Each core has two hardware threads and 256KiB of L2 cache, giving a total of 16 hardware threads and 2MiB L2 cache. Each processor also has 8MiB of L3 cache and an independent memory bus, creating a non-uniform memory architecture when both cores from both processors are used. For all tests the workstation ran Linux 3.2.0. Where appropriate, the `taskset` utility and runtime flags were used to restrict the cores on which benchmarks ran. Unless otherwise noted only one hardware thread was used per core, and cores in the same processor were selected in preference to cores in separate processors. For example, benchmark results for five cores will involve four cores from one processor and one core from the other processor.

Comparison of results was performed by close reimplementation of the benchmarks using multiple languages and concurrency frameworks:

- *CCSP C* – occam-pi runtime programmed using its C API.
- *o-pi* – occam-pi runtime programmed using occam-pi.
- *o-pi* – occam-pi runtime programmed using occam-pi compiled using LLVM (see chapter 4).
- *Erlang* – see 2.5.11 [4]. Version 5.8.5 with HiPE [200] was used.

[4]Synchronised communication is not forced, but instead the benchmark designs are modified to function with asynchronous messaging. This should be a performance benefit for Erlang.

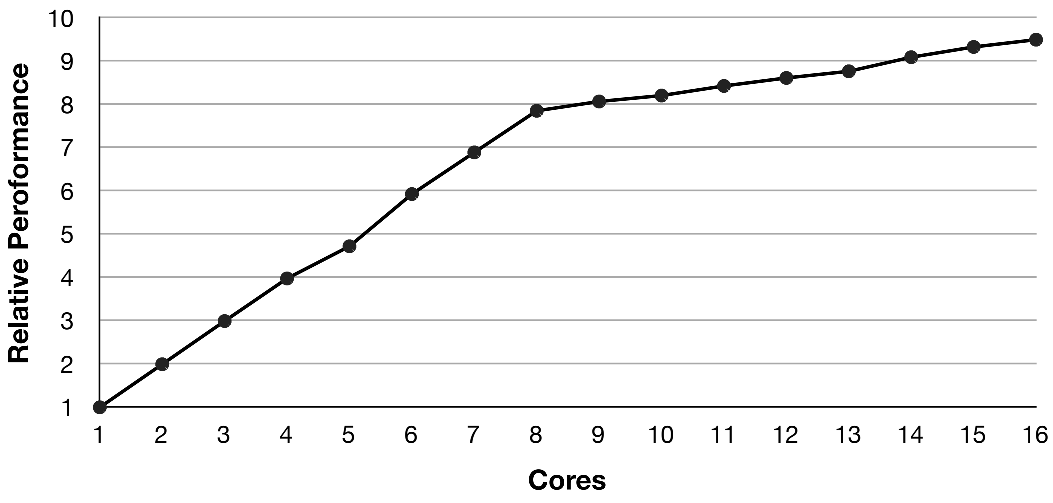


Figure 35: Scaling performance of Mandelbrot set render with increasing numbers of hardware threads.

- *Go* – see 2.5.13. Google Go version 1.1.1.
- *Haskell* – see 2.5.15. Light-weight threads and one-place buffered channels provided by the `MVar` primitive. GHC version 7.4.1 [123] was used.
- *Java* – see 2.5.17. Benchmarks use primitives from `java.util.concurrent`, in particular `ArrayBlockingQueue`. OpenJDK version 1.7.0_25 was used for compilation and execution.
- *pth C* – POSIX threads accessed via the GNU C library, see 2.5.24. Mutual exclusion (`pthread_mutex_t`) and condition variables (`pthread_cond_t`) are used to construct one-place buffered communication channels.

3.15.2 Mandelbrot

In this first benchmark the coarse-grain scalability of the runtime is tested by calculating iterations of the Mandelbrot set using a group of worker processes. The test calculates 128 frames or iterations. A central process farms out lines of each iteration to 128 worker processes. The workers return the calculated image lines to the central process.

Table 4: Communication times, calculated using process ring results.

Implementation	1-core (ns) CI 95%	8-core (ns) CI 95%
CCSP C	38 - 38	44 - 45
CCSP occam-pi	31 - 31	37 - 38
Erlang	460 - 461	478 - 484
Google Go	238 - 238	499 - 536
Haskell	382 - 386	43207 - 44907
Java	4728 - 4845	39060 - 41342
pthread C	2186 - 2195	18287 - 19473

Areas of the Mandelbrot set can be independently computed in parallel, hence execution speed should increase linearly as more processor cores are utilised. The results in figure 35 show the modified runtime scheduler performs this parallelisation correctly.

In addition to enabling all cores, the results show further scaling to all hardware threads. Surprisingly, the runtime scheduler is able to extract further parallelism from these threads without any performance decrease. This suggests the hardware threads are more computationally capable than expected.

3.15.3 Process Ring

To examine communication overheads, this test constructs a ring of *n element* processes, and one *initiator* process. Element processes loop: they receive an integer token from the previous process in the ring, increment it, then send it on to the next process. The initiator adds tokens, counts them passing and after a given count removes them from the ring. By increasing the number of tokens "in flight" around the ring, the number of potentially concurrently executing processes is increased.

Given the time taken for a single token to circulate the ring an estimate of the mean communication time of each language runtime can be computed as $time \div ((elements + 1) \times roundtrips)$. For all examples, there are 255 element processes and tokens make 1024 round trips. With 255 elements it is likely that all processes will fit within the processor caches, allowing the examination of the best-case communication time.

Table 4 shows communication times in nanoseconds. These are based on the circulation of a single token with one core or eight cores enabled.

The communication times for Erlang and the modified occam-pi runtime are relatively unaffected by the number of processor cores. While both CCSP C and CCSP occam-pi implementations use the same runtime, the occam-pi compiler caches scheduling pointers in registers, reducing the kernel call overhead. This explains the small difference in the results.

Google Go performance is impacted slightly by the addition of processor cores, but is broadly comparable to Erlang. Java and POSIX threads see an order of magnitude slow down with the addition of processor cores. Haskell performance degrades significantly, two orders of magnitude, with the addition of cores. This reflects internal contention exposed by multiple processors accessing the Haskell runtime in parallel.

The plot in figure 36 shows the time taken for 1024 circulations of 64 concurrent tokens as the number of processor cores is increased. With the exception of POSIX threads and POSIX threads-based Java, all the implementations show decreased performance with increasing numbers of cores. This reflects the fact that, for user processes, communicating between processor cores is more expensive than communication on the same core. As the number of concurrent processes increases, they are scheduled on to separate cores, increasing the communication costs. In particular, the NUMA aspects of the system beyond four cores show up as a significant degradation of performance.

POSIX threads and Java performance are noticeably improved by more cores. This then improves performance as interprocessor communication via processor caches is faster than Linux's context-switch.

While Erlang performance is stable, Haskell performance notably degrades with increasing numbers of cores.

The modified occam-pi runtime, while not performing as in the optimal case (single-core), does control the slow down with increasing numbers of cores. Performance is not expected to degrade below interprocessor communication time.

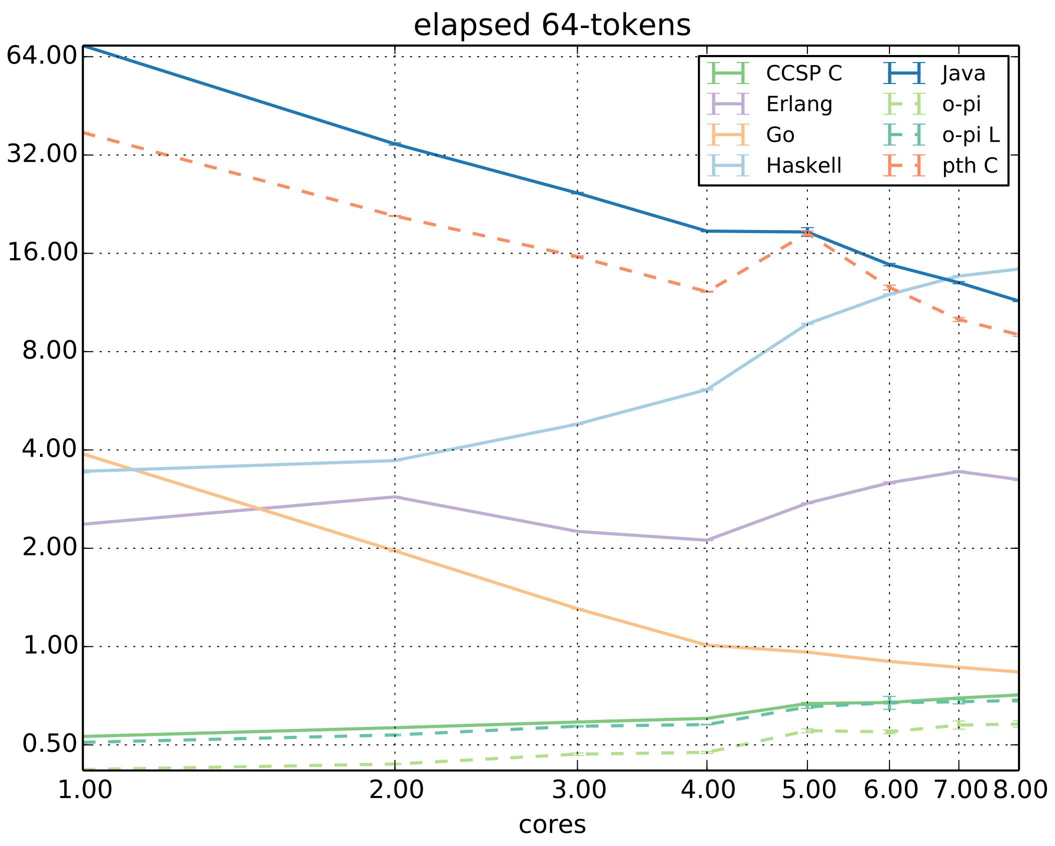


Figure 36: With 64 tokens in the process ring, increasing numbers of processor cores.

3.15.4 Agent Simulation

As previously mentioned, occam-pi is being used for complex systems modelling as part of the CoSMoS project [3]. The investigators are exploring using process-oriented methodologies for building models of emergent behaviour, and creating a generic toolkit for doing so. One of the early models investigated by the group was a process-oriented implementation of Craig Reynolds' *boids*, a simulation of flocking behaviour [213]. The CoSMoS project's implementation, *occoids*, employs *agent* processes with internal concurrency to implement the boids and their behaviour rules [35]. Agent processes move through a grid of *location* processes, connecting and reconnecting as they go. The topology of space can be modified by adjusting the underlying network connections, and this technique has been exploited to build an implementation which spans a network

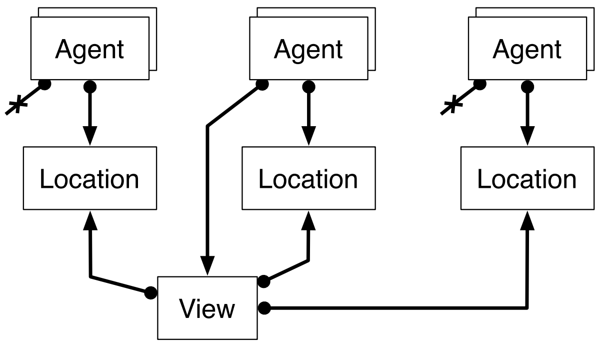


Figure 37: Simplified occoids process network. Boxes represent concurrent processes. Arrows represent two-way client-server channel connections, with the arrow pointing at the server. Agent processes connect to their present location, and "see" other agents via the location's view.

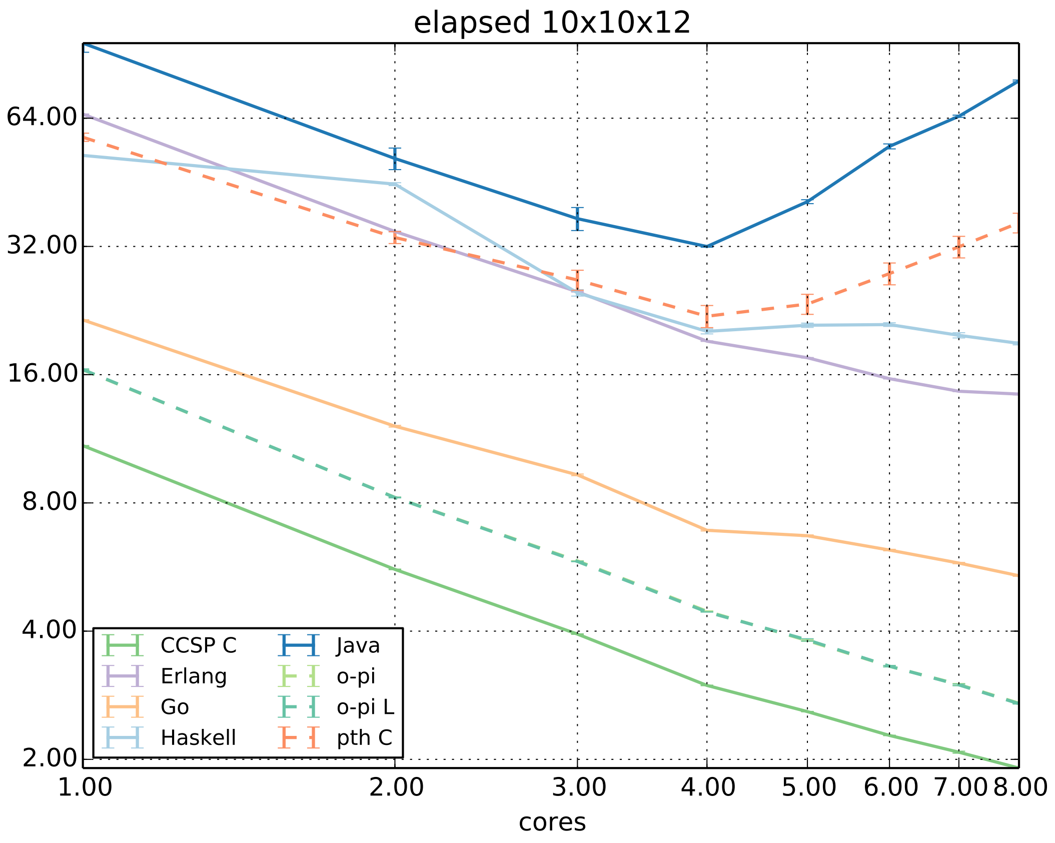


Figure 38: Increasing the number of cores applied to the agent simulation. The simulation is a 10x10 grid with 1200 agent processes.

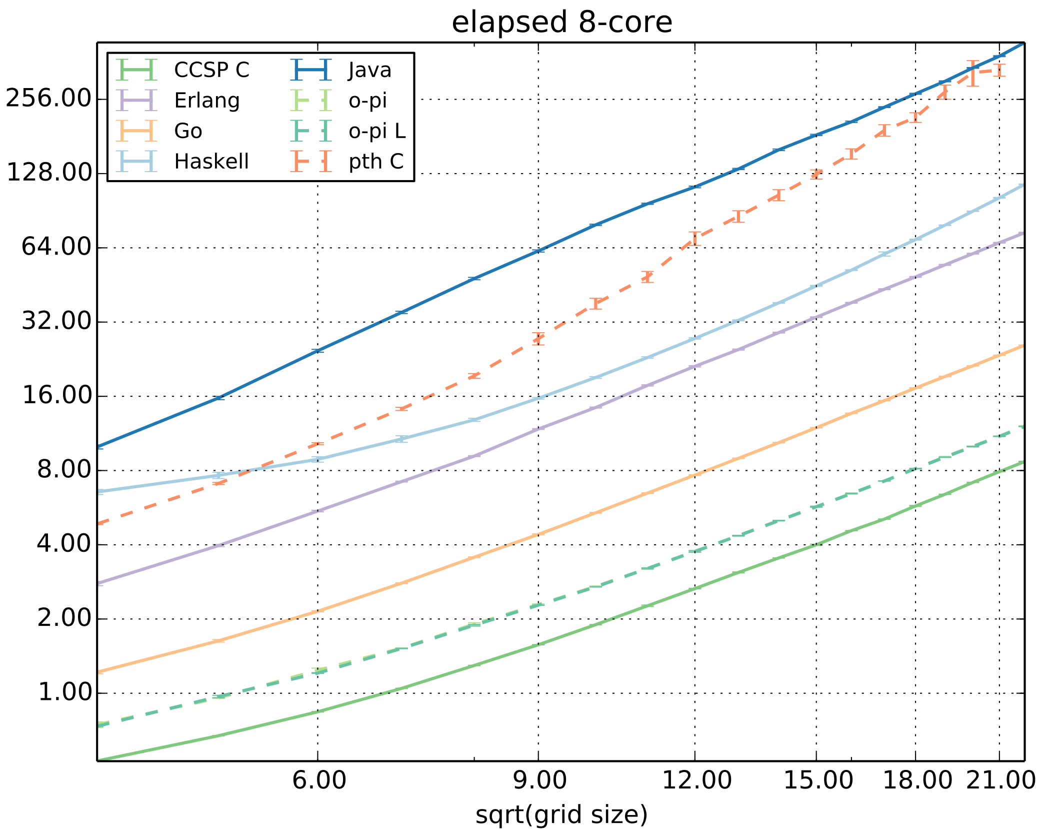


Figure 39: Simulation time for agents benchmark with increasing grid size. Each grid location has 12 initial agents. The x-axis is the number of locations in each axis.

of computers with only minor changes to the code base.

This section uses a benchmark constructed to mimic the behaviour of occoids. This benchmark is designed to be easy to implement in other languages, and produces results which allow the verification of an implementation's correctness. The simulated space is a two-dimensional torus and agent positions are represented as integers relative to the centre of their present location. The occoids simulation uses floating-point variables so as not to unduly quantise space; however, integers allow reliable verification of the simulation output and avoid any associated variations in floating-point support.

With reference to the process diagram in figure 37, location processes, acting as servers, maintain a data structure containing all agents presently in their grid area. View processes act as servers to clients, but also as clients to the location processes, building aggregate lists of all agents within nine adjacent locations each simulation step. Agent processes query a *view* process, and calculate a repulsive force from other visible agents, applying an internal *bias*. Having determined the force, the agent signals movement to its location, reconnecting to a new location if appropriate. Agents maintain a consistent time step using barrier synchronisations between activity phases.

The bias is updated based on the position of the agent and the number of other agents seen. In effect the bias produces randomised behaviour in the agents. The initial position of all other agents in the simulation acts as the seed, and hence can be easily reproduced.

As a comparison to the process-oriented design, a hand-optimised data parallel version using POSIX threads was implemented. Only one thread is used per processor core and each thread executes a fixed number of agents. Data updates are performed in parallel using fine-grain locking of location data structures. This version represents the optimal case and appears as *pthread DP C* in figures 38 and 39.

Figure 38 shows comparative results with increasing numbers of available processor cores with a fixed-size world grid and number of agents. With reference to the process-oriented implementations, the modified occam-pi runtime provides a marked improvement in performance and scalability. Haskell fails to achieve more than 50% speed up, even with eight available processors. POSIX threads achieves approximately a 150% speed up over eight cores, while the occam-pi runtime achieves 500%. Scalability of the occam-pi runtime also outstrips the ideal POSIX threads solution over eight cores.

Again, the NUMA nature of the architecture changes the performance profile of the machine after four cores. This negatively effects the performance of all implementations except those using the occam-pi runtime. This may be due to the heavy optimisation for cache locality.

The overall performance of the C version using the occam-pi runtime is 50% of the optimal case. Assuming this performance loss is communication and scheduling overhead then further refinements of the CCSP scheduler and compiler integration should be able to bring performance closer to the optimal case. The reduced performance of occam-pi compared to C is due to more efficient optimisation of serial code by the GNU C compiler than the occam-pi compiler. Techniques for overcoming this using LLVM assembly are discussed in chapter 4.

Figure 39 shows results when scaling the simulation size with eight cores. Simulation size is controlled by increasing the grid size and number of agents. In this test the occam-pi runtime also outperforms other process-oriented implementations. The other process-oriented implementations increasingly diverge from the optimal case with increasing problem size. The similarity of the occam-pi runtime's scaling curve to the optimal suggests that refinement of this design may be sufficient to achieve near optimal performance.

3.15.5 Performance Counters

To evaluate the underlying reasons for the performance shown in section 3.15.3 and 3.15.4 this section presents data gathered from hardware performance counters [144]. Of particular interest is whether the batch mechanism and associated heuristics are improving cache utilisation. To assess this, results for the following counters are shown:

- *L1 dcache loads*: the rate of loads from the L1 data cache.
- *L1 dcache load misses*: the rate of cache misses when loading from the L1 data cache.
- *L1 dcache stores*: the rate of stores to the L1 data cache.
- *L1 dcache store misses*: the rate of cache misses when storing to the L1 data cache.
- *LLC loads*: the rate of loads from the last-level cache (L3 in the test system).
- *LLC load misses*: the rate of cache misses loading from the last-level cache.
- *LLC stores*: the rate of stores to the last-level cache (L3 in the test system).
- *LLC store misses*: the rate of cache misses storing to the last-level cache.

For the counters used, a low rate of cache misses is desirable. Fewer cache misses mean the given cache is being used more efficiently. This assumes that memory accesses are occurring. Hence the rate of loads or stores to a given cache is also presented.

Figures 40, 41, 42 and 43 show counters for the process ring benchmark as in section 3.15.3. There are 64-tokens circulating a ring of 1024 processes with increasing numbers of active processor cores. In all cases results for the CCSP scheduler show significantly lower rates of cache misses; despite, in many cases, having higher rates of cache access. The lower rates of last-level cache access shown in figures 42 and 43 are reflective of higher hit rates in L1 and L2 caches. It is also worth noting that beyond four cores inter-processor communication significantly reduces the efficiency of the last-level cache for all languages.

Figures 44, 45, 46 and 47 show counters for the agents benchmark as in section 3.15.4. A fixed-size grid is used with increasing numbers of active processor cores. At L1 the rate of cache loads and stores of C compiled code with the CCSP scheduler is broadly similar to other languages. Code output from the occam-pi compiler is less efficient and generates more loads and stores. While L1 cache miss rates are comparable to languages such as Google Go and Erlang they are more stable with the addition of processor cores. In particular, CCSP scheduler code has a lower rate of L1 store misses.

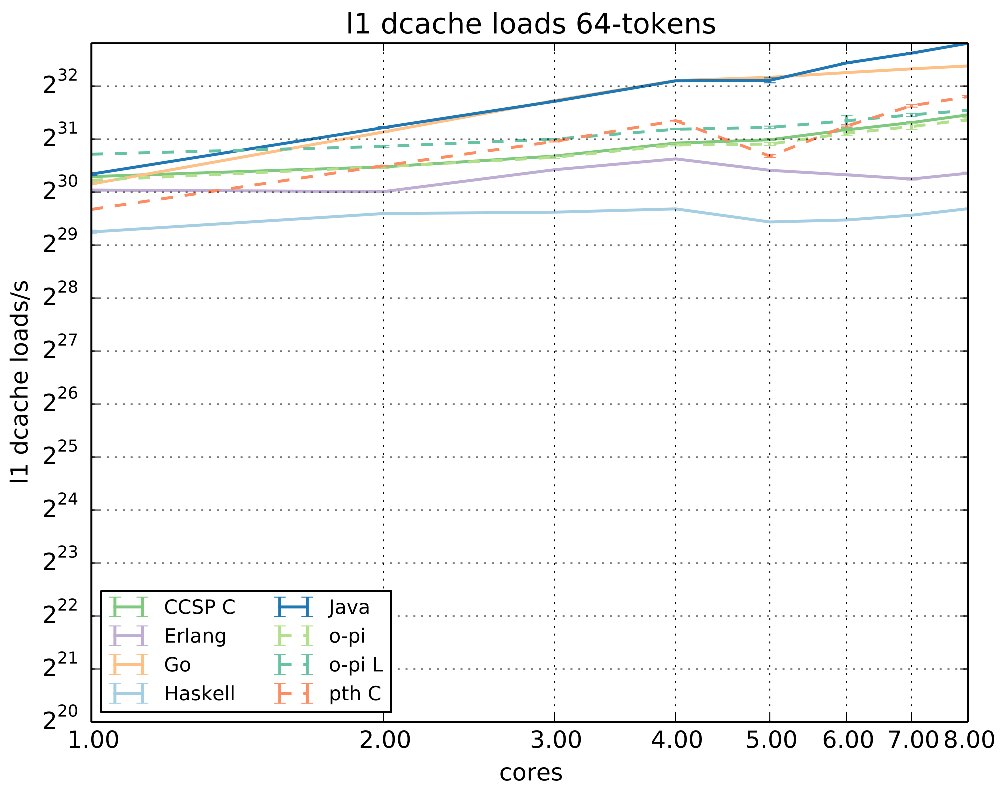
Java and POSIX threads access rates decrease as synchronisation traffic impacts overall performance.

At the last-level of cache all code using the CCSP scheduler is producing significantly lower rates of loads, stores and associated cache misses. This indicates the efficiency of L2 cache has been significantly improved (as last-level is L3). Again a phase shift in last-level misses is observable at five or more processor cores.

Figures 48, 49, 50 and 51 also show counters for the agents benchmark. In this case the number of processor cores is kept constant at eight and the grid size increased. For these results a constant rate of loads, stores and associated misses is expected. Constant rates indicate the cache is being used consistently independent of grid size. At L1 the results broadly fit this pattern; however, at large grid sizes the rate of load misses for the CCSP scheduler increases bring it into line with the performance of Google Go or Erlang. Java and POSIX threads miss rates decrease with larger grid sizes. This a reflection of reduced overall performance (more time spend context switching and synchronising).

At the last-level of cache code using the CCSP scheduler is consistently out performing comparable languages such as Google Go and Erlang. This reinforces earlier observations of improved L2 cache performance. Java and POSIX threads have better last-level cache performance than the CCSP scheduler. This performance is probably indicative of a large amount of time spent in highly optimised process switching (kernel) code and little time spent on benchmark execution as both have one or two orders of magnitude worse performance with respect to benchmark execution time.

Overall the agents benchmark results show an approximate improvement in cache utilisation by the CCSP scheduler of two to four times over comparable languages such as Google Go and Erlang. This is directly comparable to the two to four times faster execution of benchmark code in comparison to those languages.



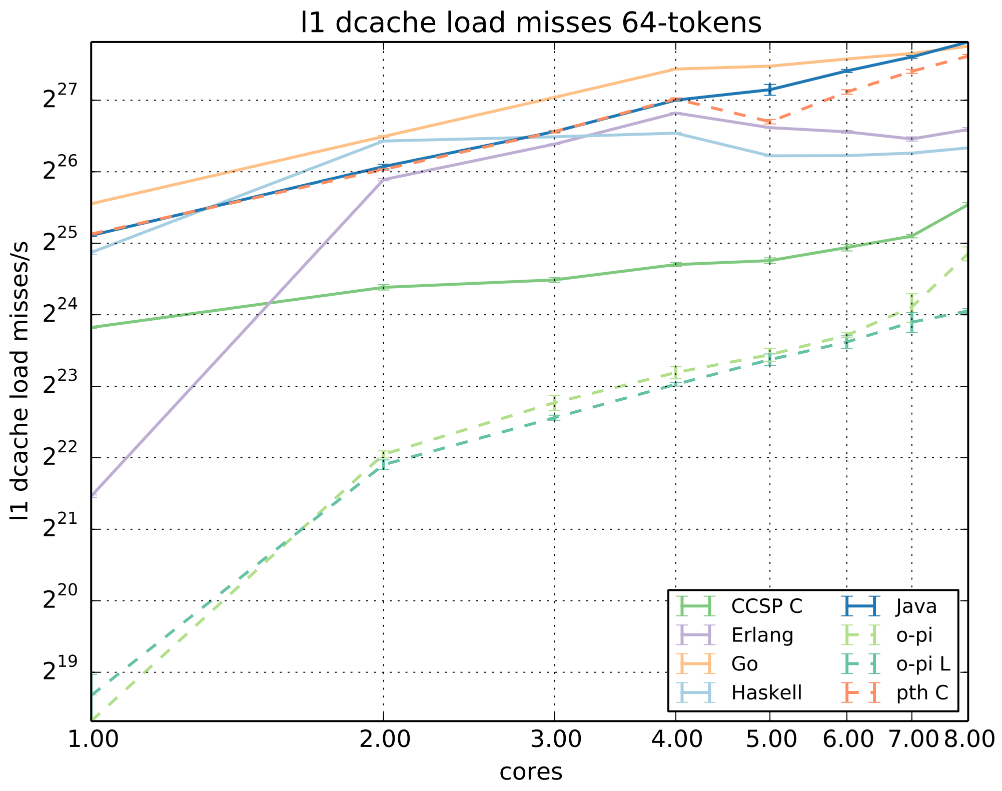
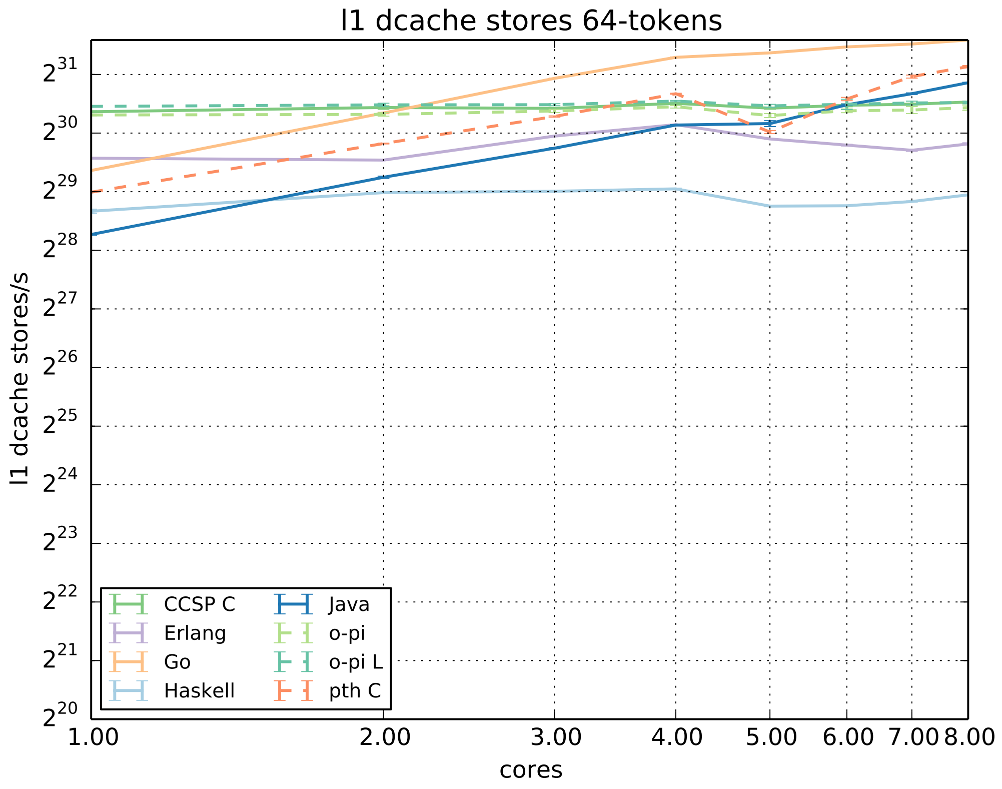


Figure 40: L1 data cache loads and load misses with 1024 element process ring and 64 tokens.



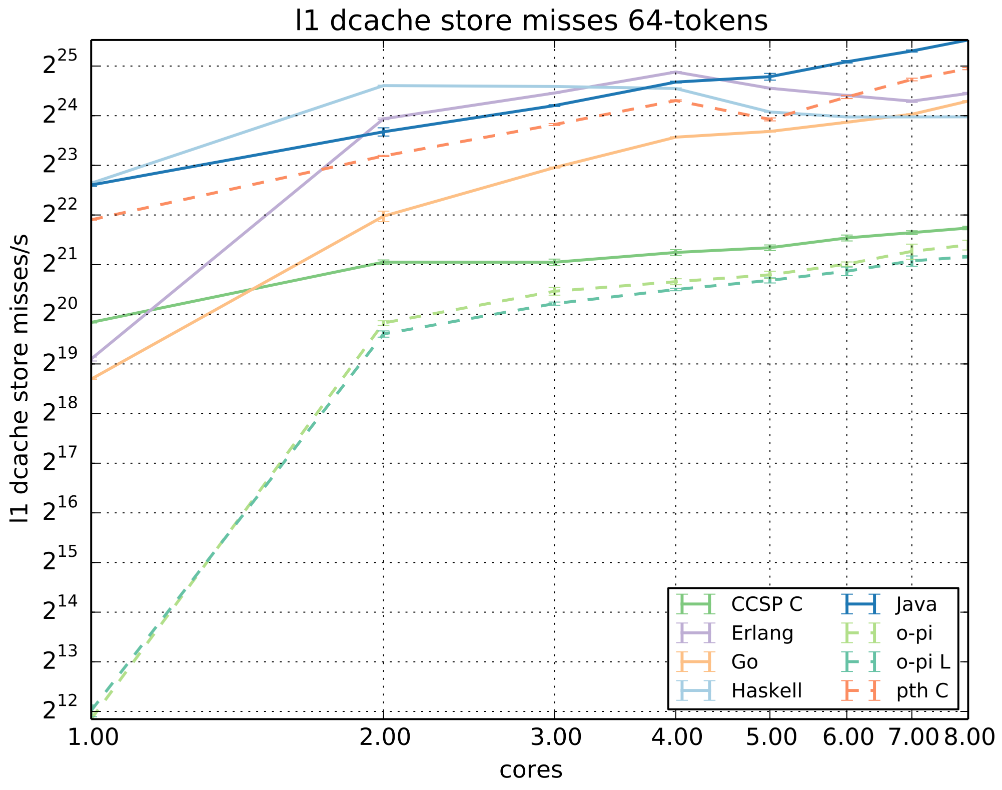
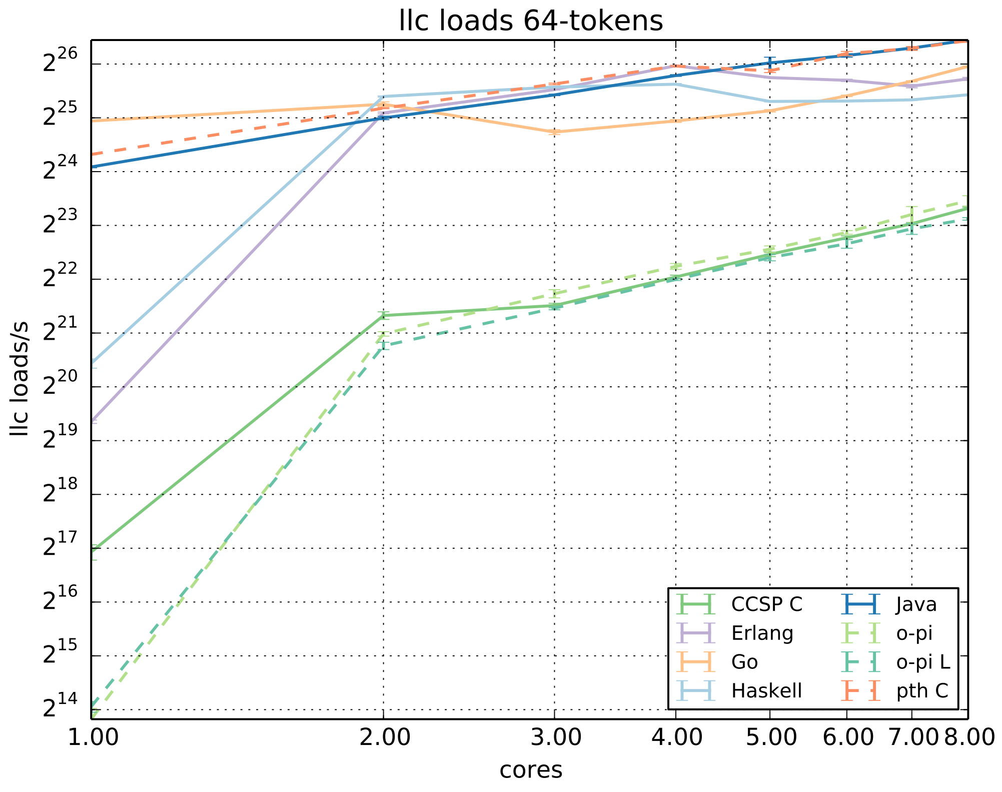


Figure 41: L1 data cache stores and store misses with 1024 element process ring and 64 tokens.



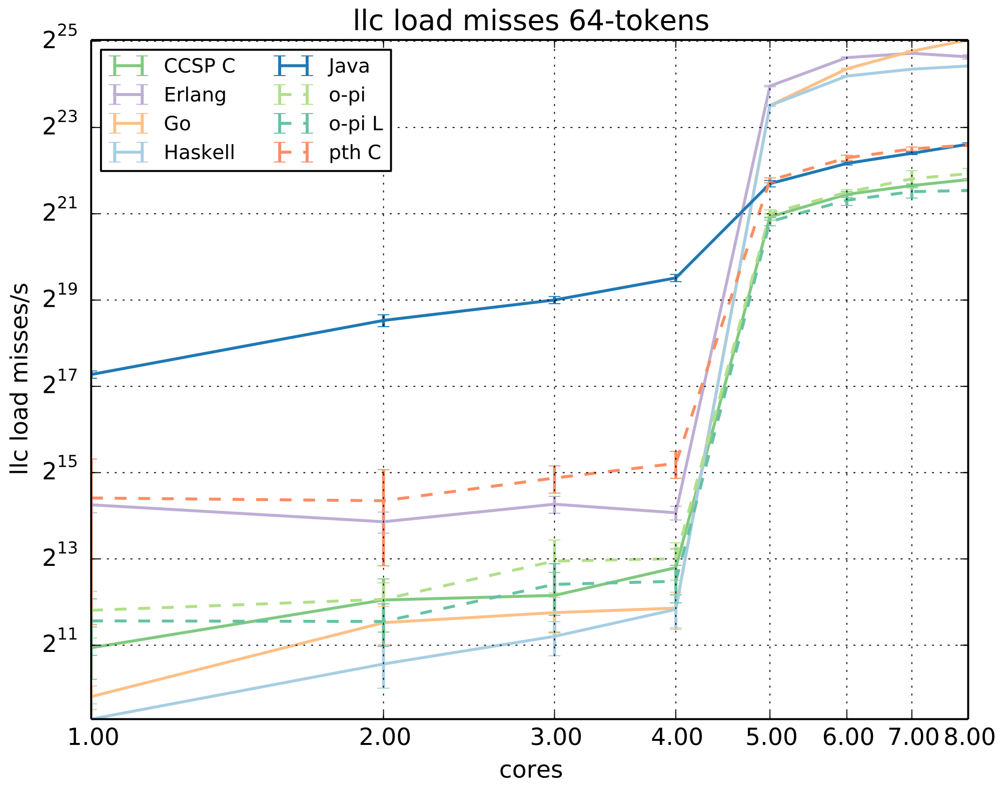
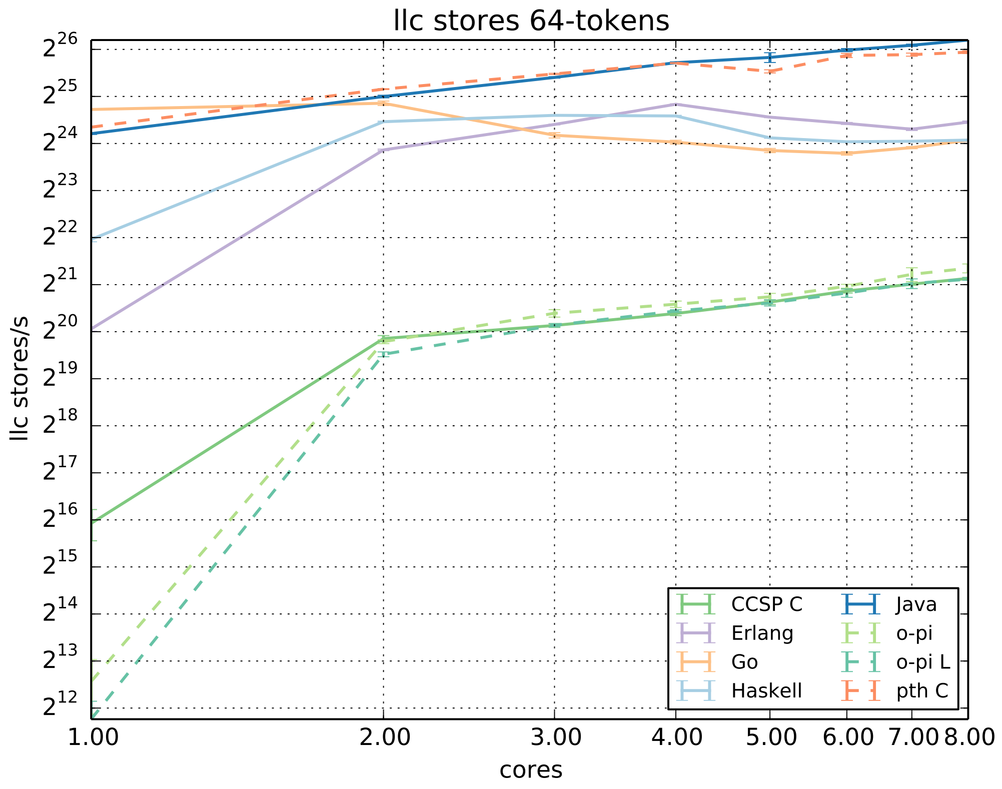


Figure 42: Last-level cache loads and load misses with 1024 element process ring and 64 tokens.



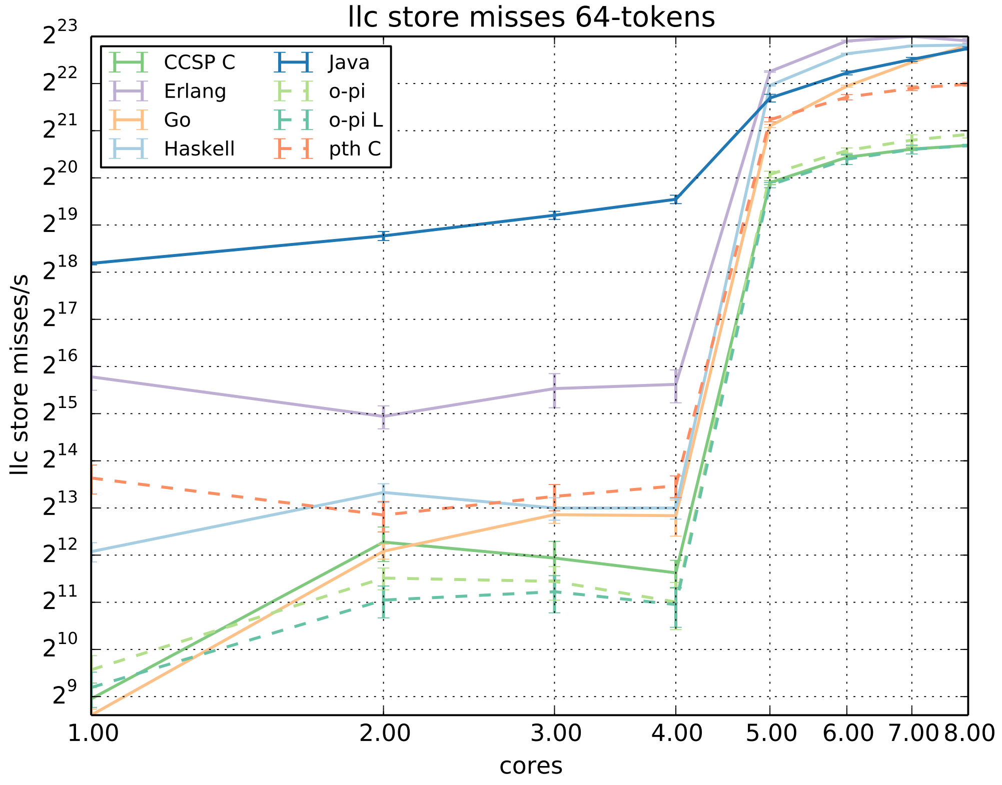
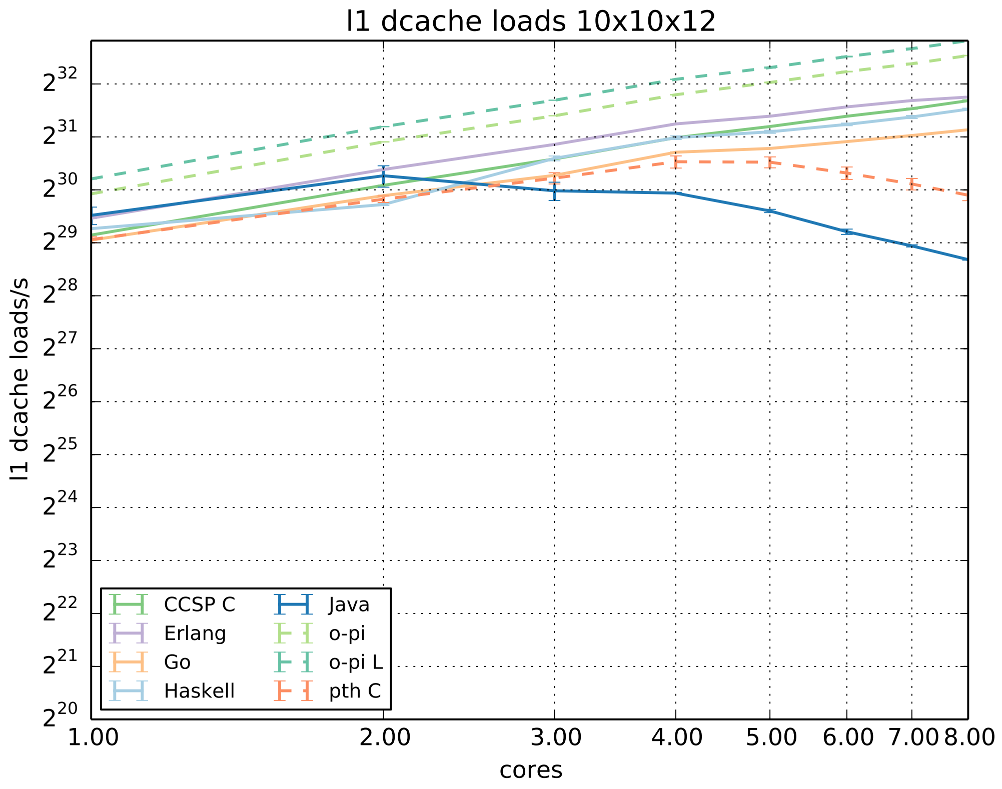


Figure 43: Last-level cache stores and store misses with 1024 element process ring and 64 tokens.



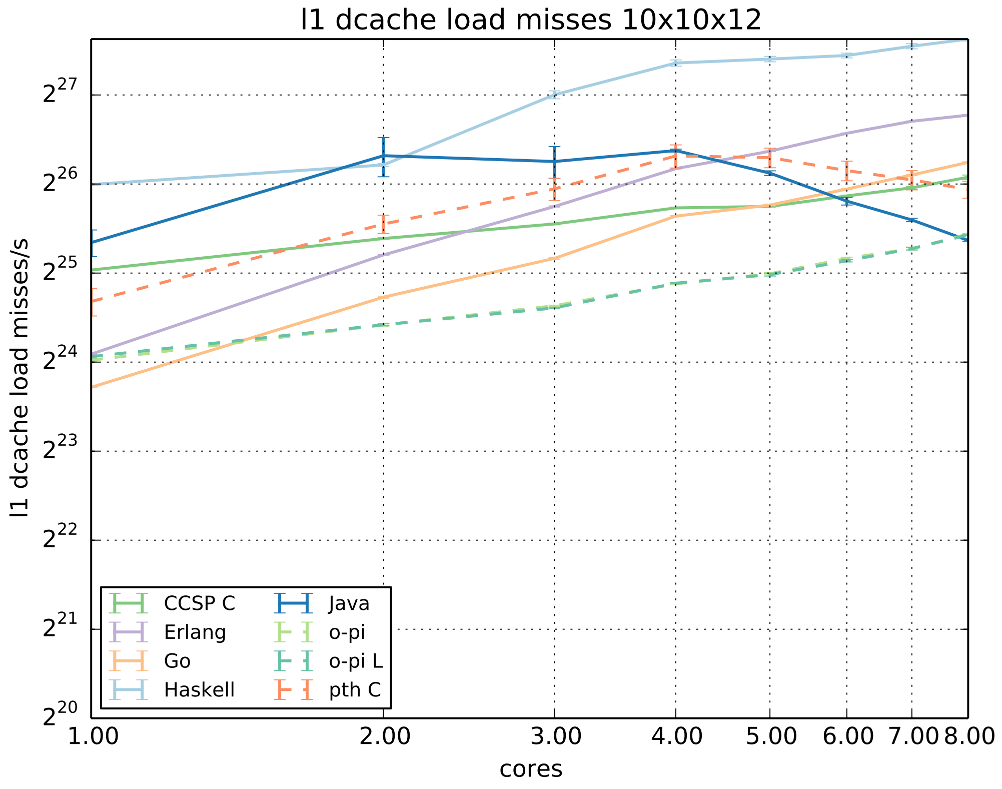
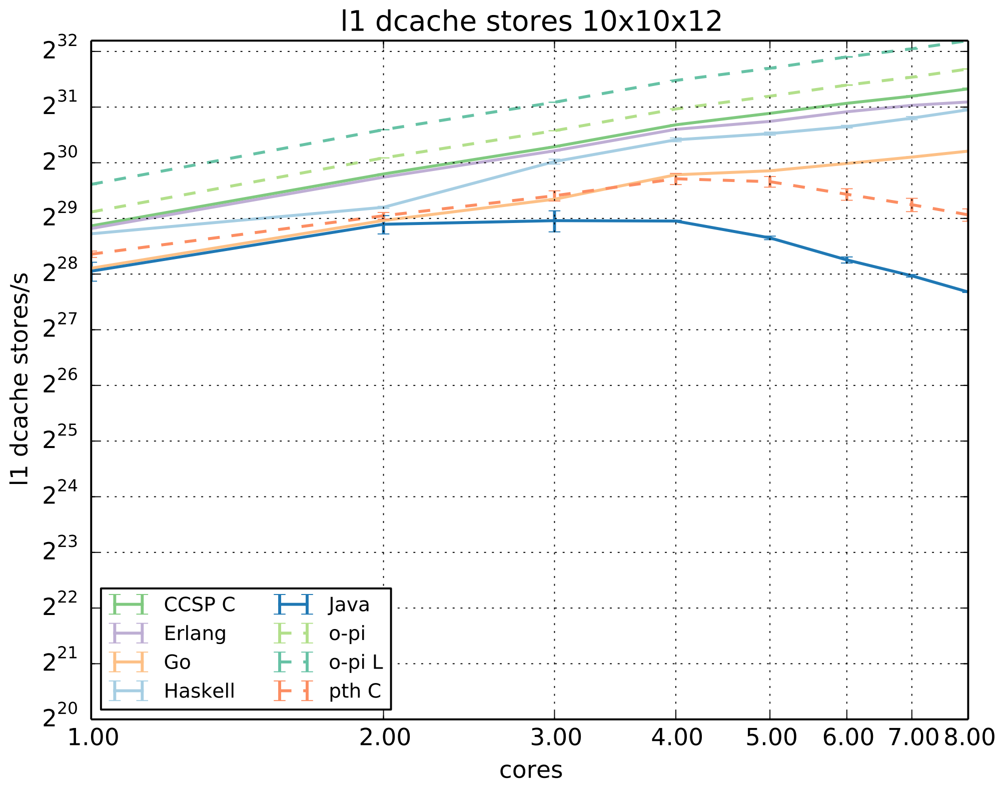


Figure 44: L1 data cache loads and load misses for agents benchmark with fixed grid size and increasing numbers of active processor cores.



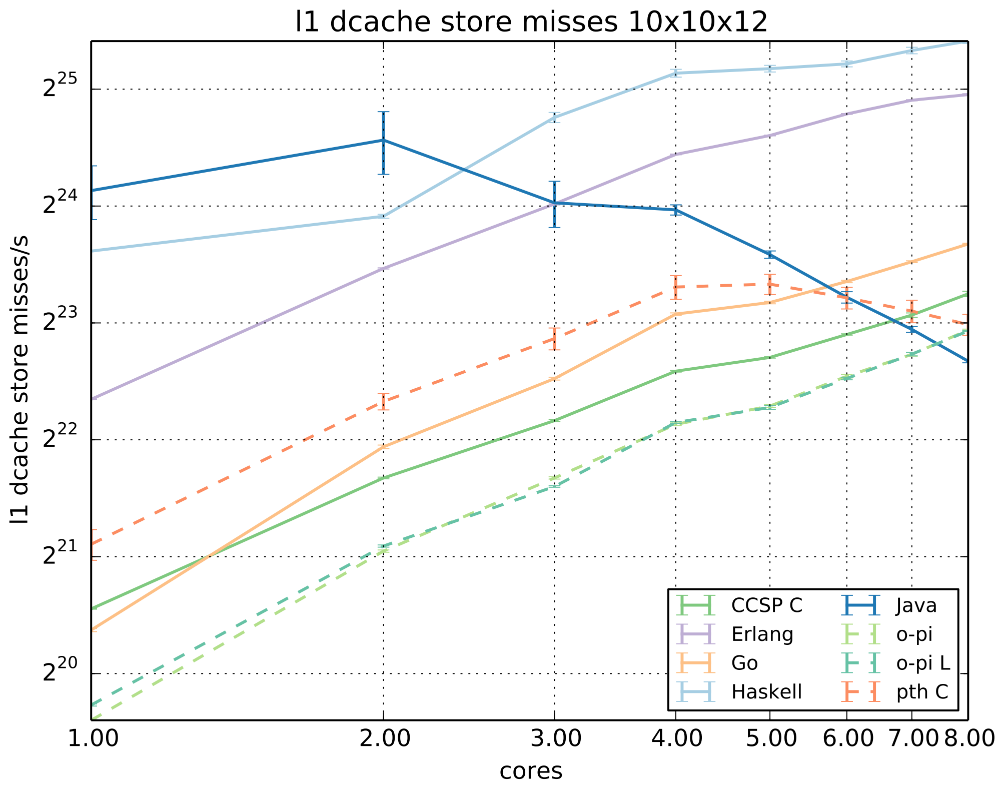
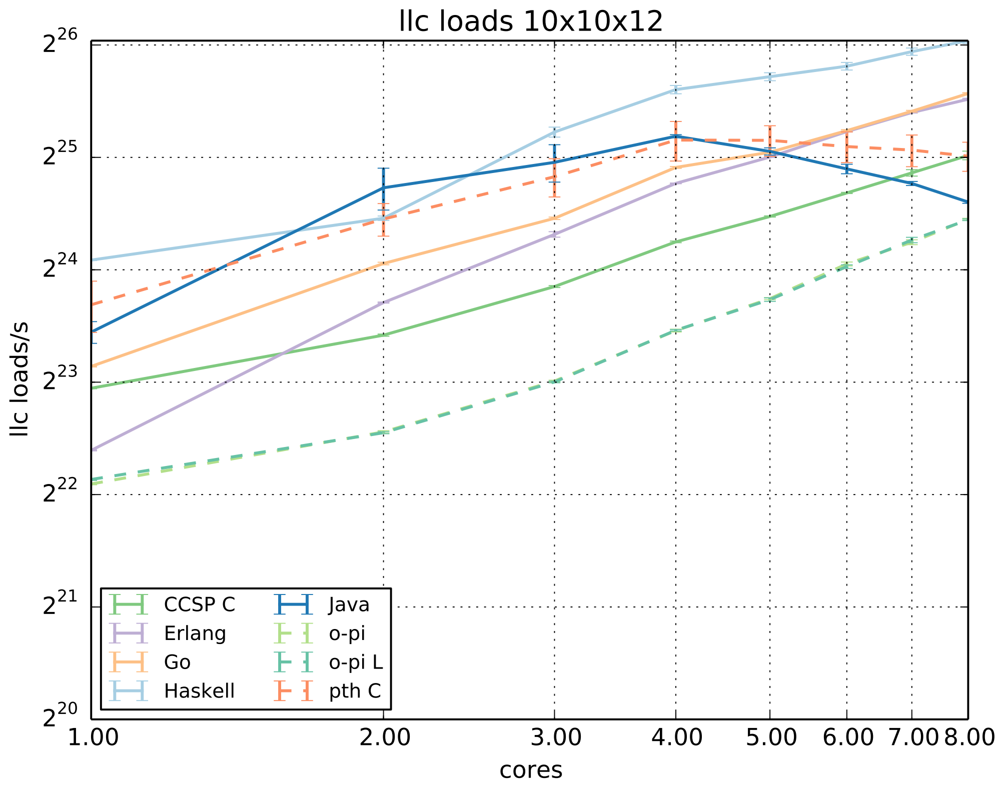


Figure 45: L1 data cache stores and store misses for agents benchmark with fixed grid size and increasing numbers of active processor cores.



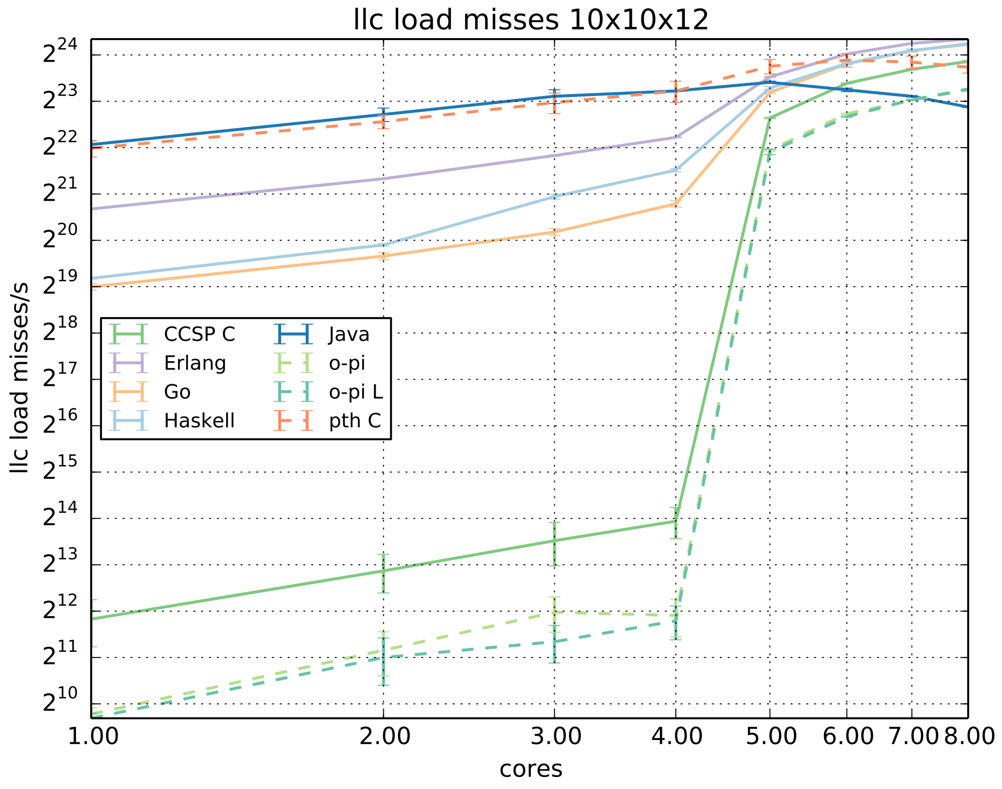
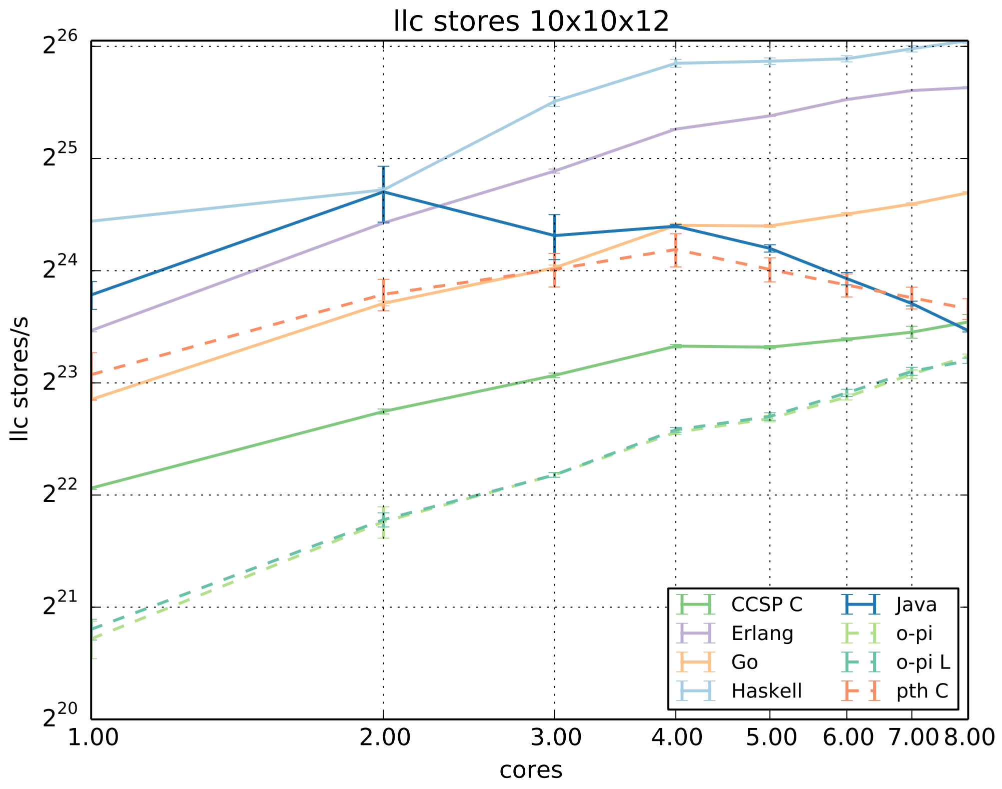


Figure 46: Last-level cache loads and load misses for agents benchmark with fixed grid size and increasing numbers of active processor cores.



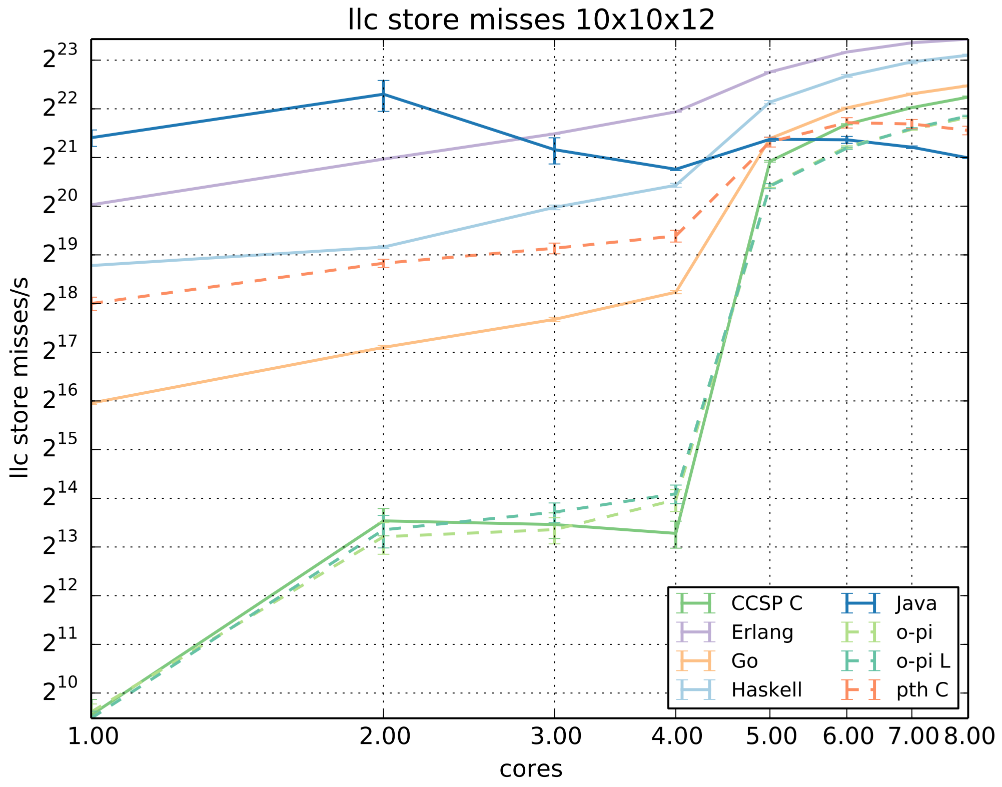
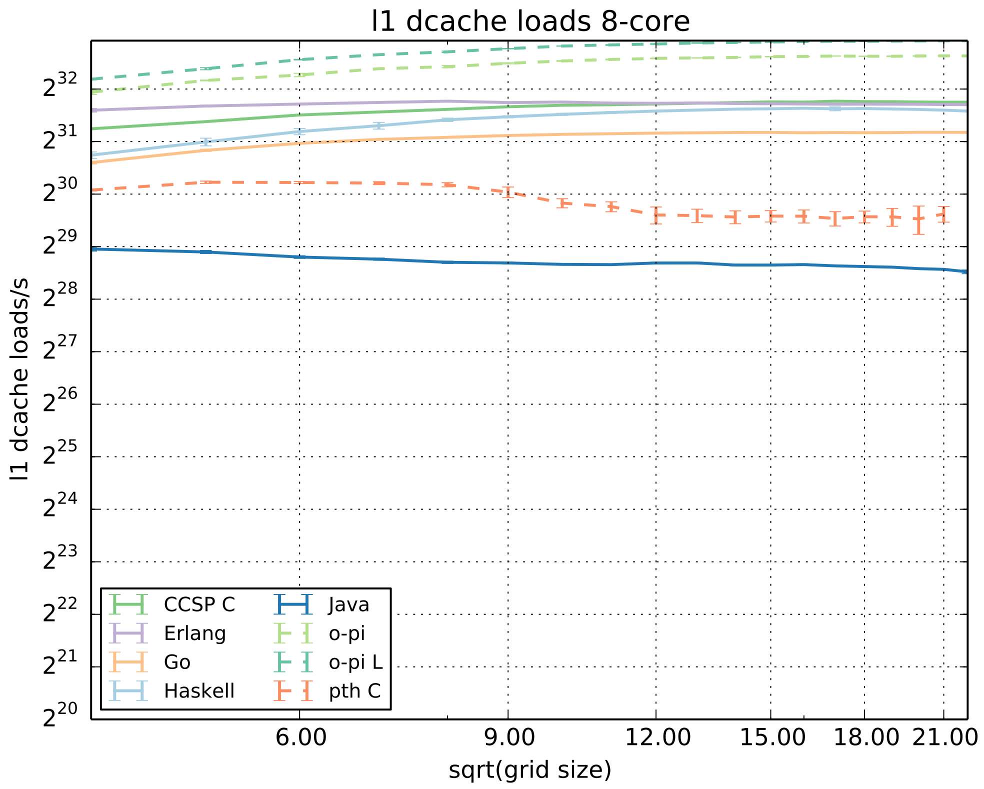


Figure 47: Last-level cache stores and store misses for agents benchmark with fixed grid size and increasing numbers of active processor cores.



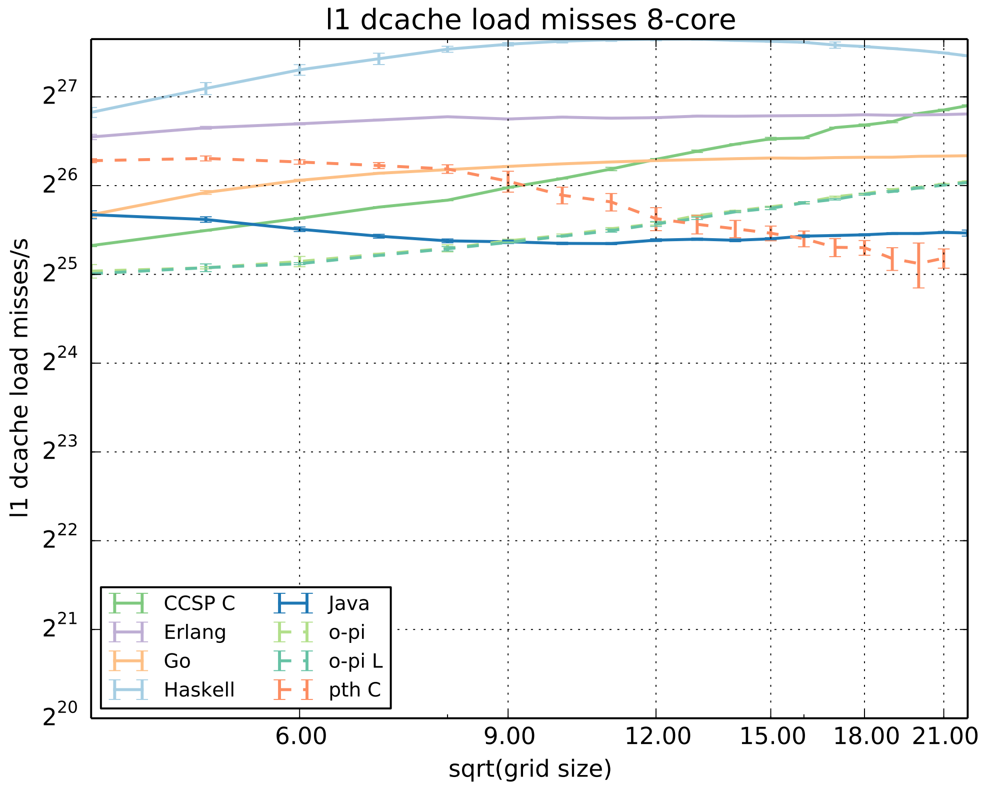
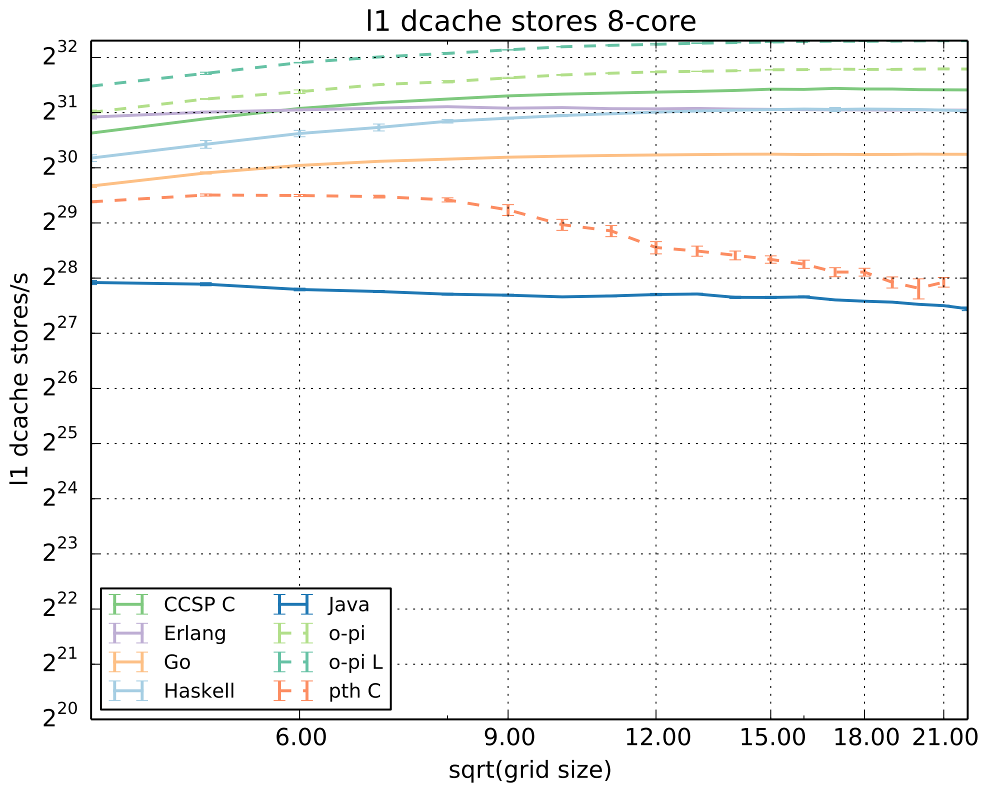


Figure 48: L1 data cache loads and load misses for agents benchmark with a variable grid size and eight active processor cores.



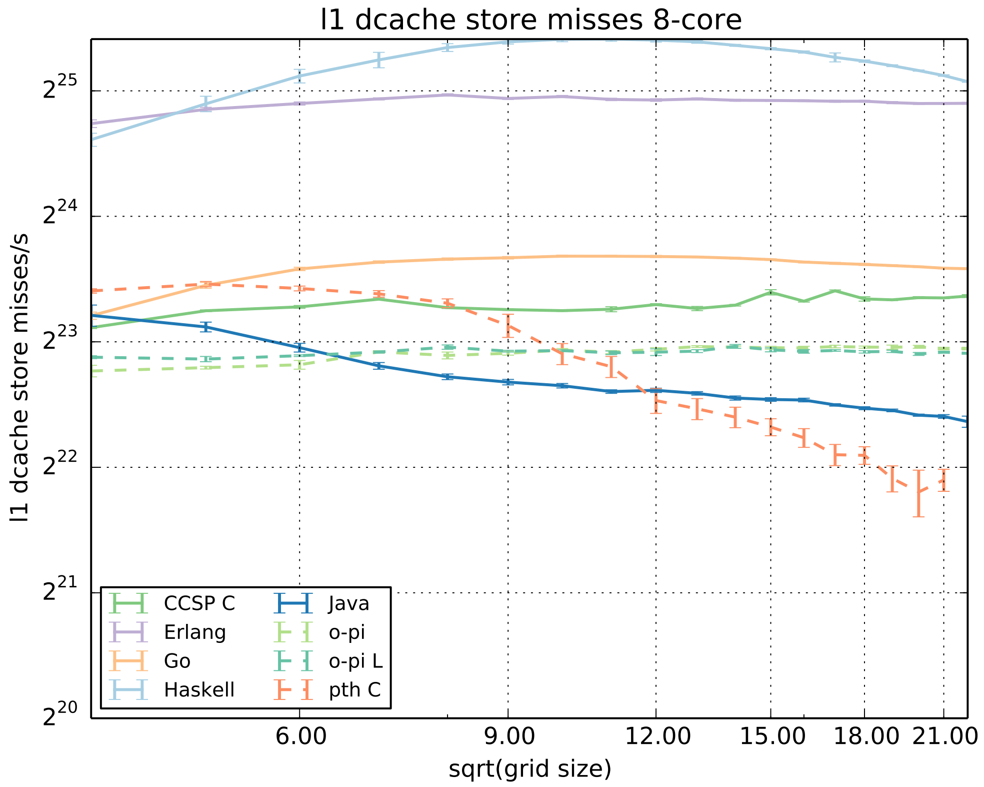
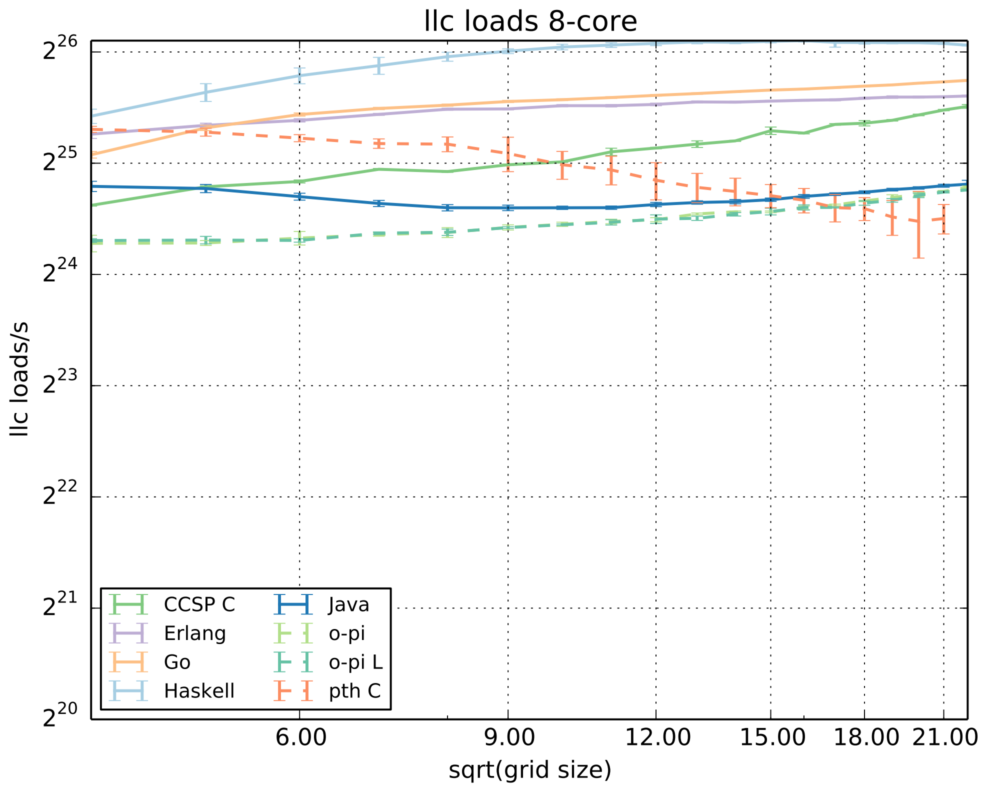


Figure 49: L1 data cache stores and store misses for agents benchmark with a variable grid size and eight active processor cores.



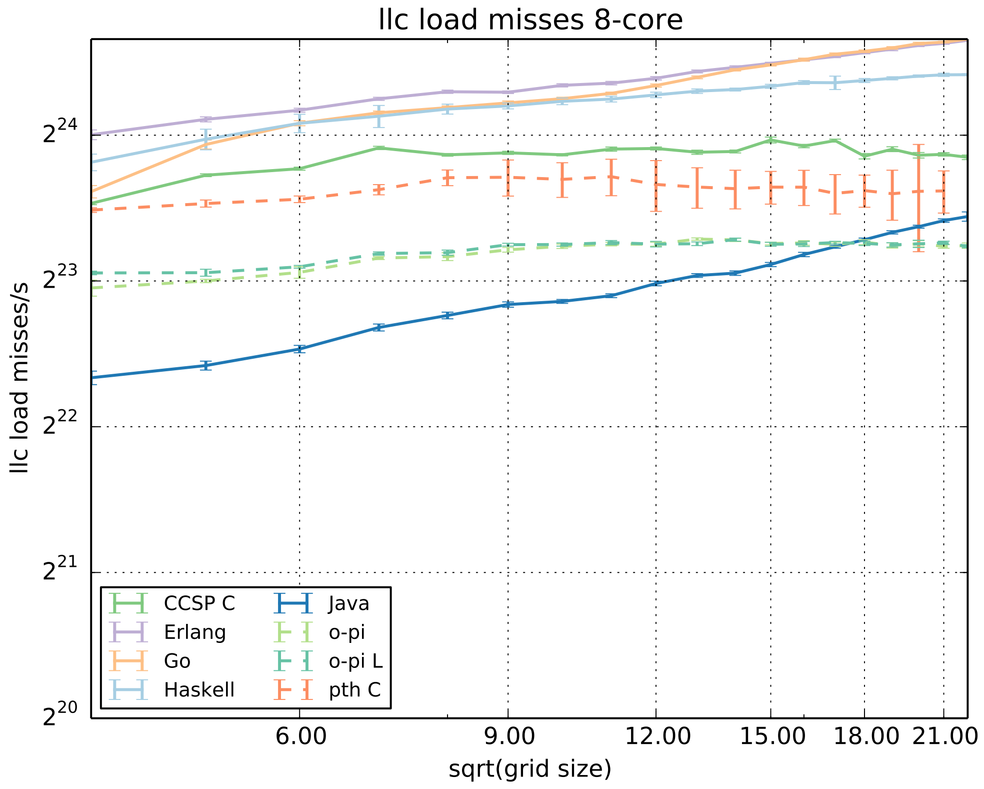
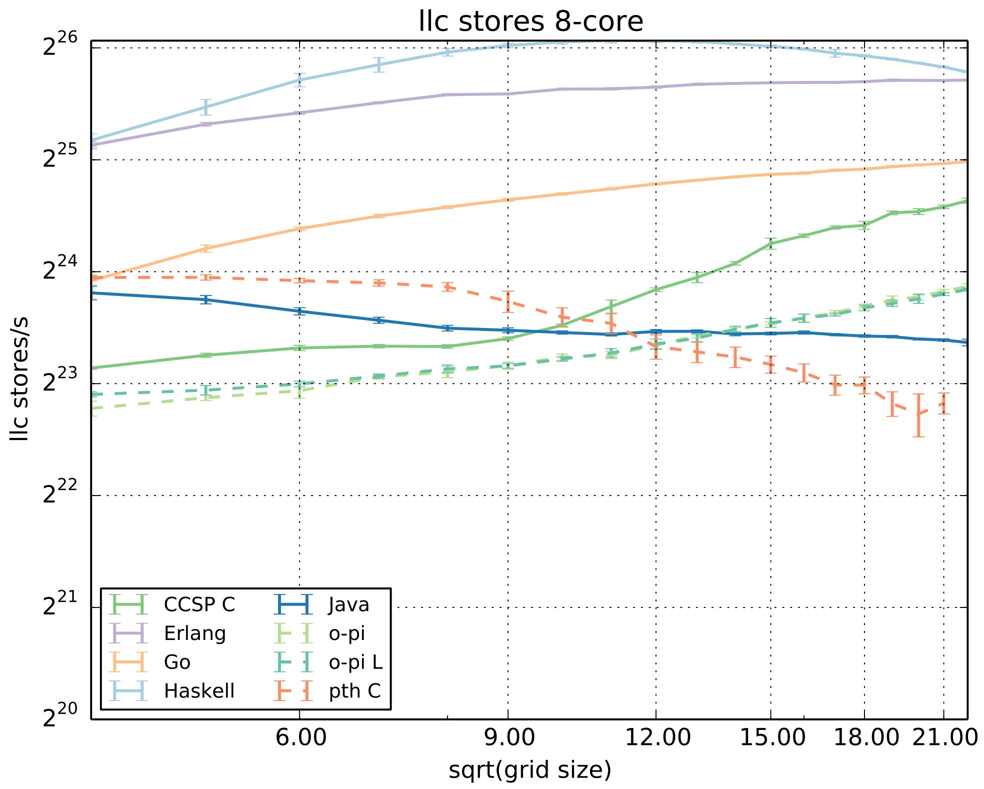


Figure 50: Last-level cache loads and load misses for agents benchmark with a variable grid size and eight active processor cores.



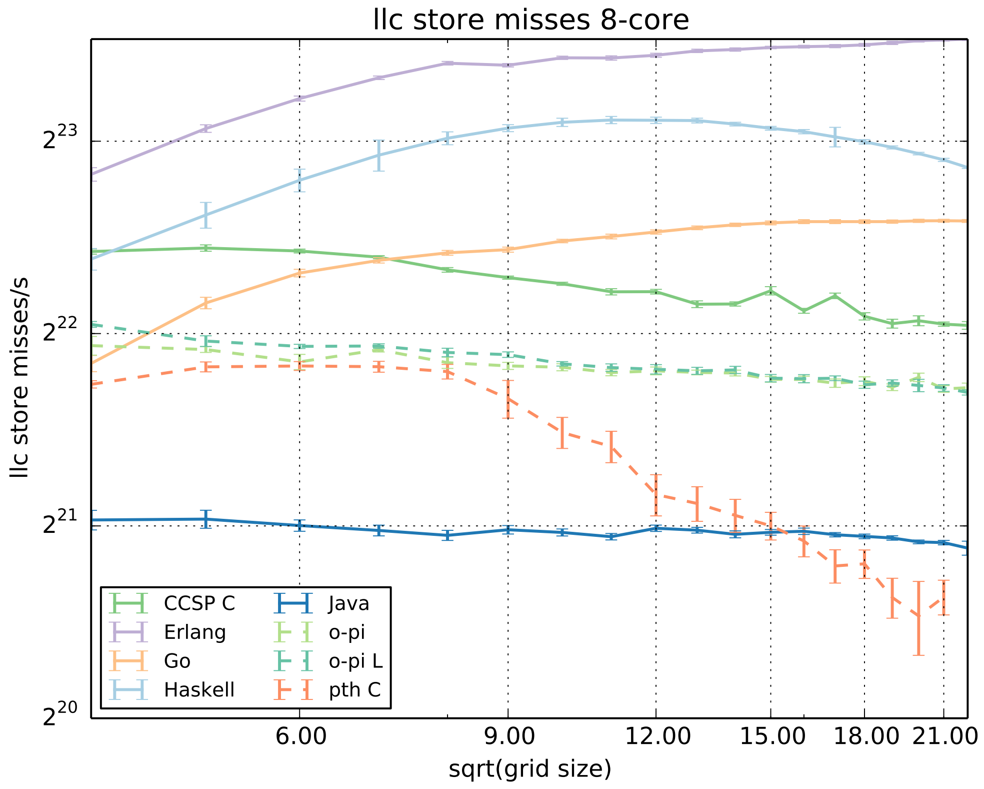


Figure 51: Last-level cache stores and store misses for agents benchmark with a variable grid size and eight active processor cores.

3.16 Conclusions

A multicore scheduler for fine-grain concurrent software developed using process-oriented programming was designed and implemented. Process-oriented designs have a high degree of interprocess communication and involve many more processes than physical processors. This is addressed in the runtime design by ensuring that:

- The serialisation bottleneck of a global run-queue is avoided by scheduling processes independently on each core.
- Cache utilisation is improved by batching communicating processes.
- No programmer intervention is required to achieve multicore execution of process-oriented designs. Processes and batches are automatically distributed and migrated between processor cores.
- Contention within the scheduler is reduced using lock-free and wait-free algorithms.
- Lock-free algorithm performance is optimised by minimising the number of atomic instructions, particularly in hot paths.

The performance results presented here show that by addressing these points a modified occam-pi runtime has significantly better performance than a number of other frameworks for implementing process-oriented designs. Specially, the occam-pi runtime brings the performance of process-oriented software close to that of optimised multithreaded implementations. Using this runtime design, process-oriented design can be applied to develop software for multicore systems without the associated complexities and hazards of threads, locks and shared-memory. Furthermore, refinements to this design should allow unmodified process-oriented software to fully utilise hardware parallelism in future generations of multicore processors (chapter 6) [176, 24, 250, 166].

Chapter 4

Compilation

This chapter presents a new intermediate representation of occam-pi programs using platform-independent LLVM assembly and continuation passing style. The LLVM assembly is derived through translation of *extended transputer code* [205]. Work in this chapter has been previously published as [217].

4.1 Motivation

The original occam language toolchain supported a single processor architecture, that of the INMOS Transputer [189, 139]. Following INMOS's decision to end development of the occam language, the sources for the compiler were released to the *occam For All* (oFA) project [204]. The oFA project modified the INMOS compiler (occ21), adding support for processor architectures other than the Transputer, and developed the basis for today's Kent Retargetable occam Compiler (KRoC) [7].

Figure 52 shows the various compilation steps for an occam or occam-pi program. The occ21 compiler generates Extended Tranputer Code (ETC) [205], which targets a virtual Transputer processor. Another tool, tranx86 [47], generates a machine object from the ETC for a target architecture. This is in turn linked with the runtime kernel

CCSP which has been modified with multiprocessor support (see chapter 3) and other system libraries.

Tools such as tranx86, octran and tranpc [261] have in the past provided support for Intel x86, MIPS, PowerPC and Sparc architectures; however, with the progressive development of new features in the occam-pi language, only Intel x86 support is well maintained. This is a consequence of the development time required to maintain support for a large number of hardware/software architectures. In recent years the Transterpreter Virtual Machine (TVM), which executes linked ETC bytecode directly, has provided an environment for executing occam-pi programs on architectures other than Intel x86 [147, 146]. This has been possible due to the small size of the TVM codebase, and its implementation in architecture independent ANSI C. Portability and maintainability are gained at the sacrifice of execution speed; a program executed in the TVM runs around 100 times slower its equivalent tranx86 generated object code.

This chapter describes a new translation for ETC bytecode, from the virtual Transputer instruction set to the LLVM virtual instruction set [158, 10]. The LLVM compiler infrastructure project provides a machine independent virtual instruction set, along with tools for its optimisation and compilation to a wide range of machine architectures. By targeting a virtual instruction set that has a well developed set of platform backends, the aim is to increase the number of platforms the existing occam-pi compiler framework can target. LLVM also provides a pass-based framework for optimisation at the assembly level, with a large number of pre-written optimisation passes (e.g. dead-code removal, constant folding, etc). Translating to the LLVM instruction set provides access to these ready-made optimisations as opposed to writing custom optimisations within the KRoC toolchain as has been done in the past [47].

The virtual instruction sets of the Java Virtual Machine (JVM) or the .NET's Common Language Runtime (CLR) have also been used as portable compilation targets [131, 229]. Unlike LLVM these instruction sets rely on a virtual machine implementation and do not provide a clear path for linking with the CCSP runtime. This was a key motivating factor in choosing LLVM over the JVM or CLR as a new portable compilation

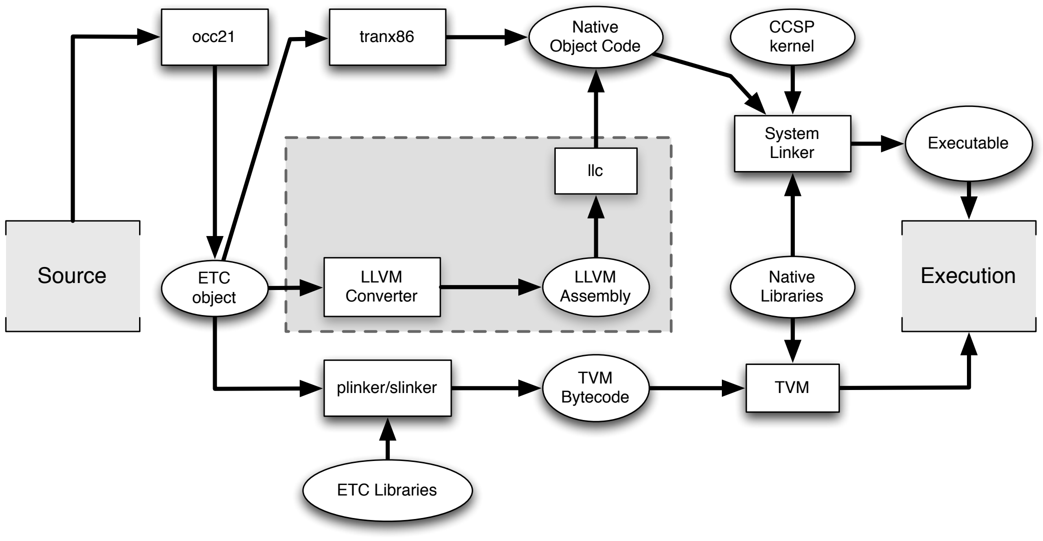


Figure 52: Flow through the KRoC and Transterpreter toolchains, from source to program execution. This paper covers developments in the grey box.

mechanism for occam-pi.

An additional concern regarding the JVM and CLR is that large parts of their code bases are concerned with language features not relevant for occam-pi, e.g. class loading or garbage collection. Given that occam-pi is desirable for programming small embedded devices (4.4.7) [146], it seems appropriate not to be encumbered with a large virtual machine. LLVM's increasing support for embedded architectures, XMOS's XCore processor in particular [173], provided a further motivation to choose it over the JVM or CLR.

4.2 LLVM

This section briefly describes the LLVM project's infrastructure and its origins. Additionally, it contains an introduction to LLVM assembly language as an aid to understanding the translation examples in later sections.

Lattner proposed the LLVM infrastructure as a means of allowing optimisation of a program not just at compile time, but throughout its lifetime [158]. This includes

```
define i32 @cube (i32 %x) {
        %sq = mul i32 %x, %x   ; multiply x by x
        %cu = mul i32 %sq, %x  ; multiply sq by x
        ret i32 %cu            ; return cu
}
```

Figure 53: Example LLVM function which raises a value to the power of three.

optimisation at compile time, link time, runtime and offline optimisation. Offline optimisations may tailor a program for a specific system, or perhaps apply profiling data collected from previous executions.

The LLVM infrastructure consists of a virtual instruction set, a bytecode format for the instruction set, front-ends which generate bytecode from sources (including assembly), a virtual machine and native code generators for the bytecode. Having compiled a program to LLVM bytecode it is then optimised before being compiled to native object code or JIT compiled in the virtual machine interpreter. Optimisation passes take bytecode (or its in-memory representation) as input, and produce bytecode as output. Each pass may modify the code or simply insert annotations to influence other passes, e.g. usage information.

The LLVM assembly language is strongly typed, and uses static single-assignment (SSA) form. It has neither machine registers nor an operand stack, rather identifiers are defined when assigned to, and this assignment may occur only once. Identifiers have global or local scope; the scope of an identifier is indicated by its initial character. The example in figure 53 shows a global function `@cube` which takes a 32-bit integer (given the local identifier `%x`), and returns it raised to the power of three. This example also highlights LLVM's type system, which requires all identifiers and expressions to have explicitly specified types.

LLVM supports the separate declaration and definition of functions: header files declare functions, which have a definition at link time. The use of explicit functions, as opposed to labels and jump instructions, frees the programmer from defining a calling convention. This in turn allows LLVM code to transparently function with the calling conventions of multiple hardware and software application binary interfaces (*ABI*s).

In addition to functions, LLVM provides a restricted form of traditional labels. It is not possible to derive the address of an LLVM label or assign a label to an identifier. Furthermore the last statement of a labelled block must be a branching instruction, either to another label or a return statement. These restrictions give LLVM optimisations a well-defined view of program control flow, but do present some interesting challenges (4.3.2).

In the examples presented here, where appropriate, LLVM syntax is commented; however, for a full definition of the LLVM assembly language please refer to the project's website and reference manual [9].

4.3 ETC to LLVM Translation

This section describes the key steps in the translation of stack-based Extended Transputer Code (ETC) to the SSA-form LLVM assembly language.

4.3.1 Stack to SSA

ETC bases its execution model on that of the Transputer, a processor with a three register stack. A small set of instructions have coded operands, but the majority consume (pop) operands from the stack and produce (push) results to it. A separate data stack called the *workspace* (3.7) provides the source or target for most load and store operations.

Blind translation from a stack machine to a register machine can be achieved by designating a register for each stack position and shuffling data between registers as operands are pushed and popped. The resulting translation is not particularly efficient as it has a large number of register-to-register copies. More importantly, this form of blind translation is not possible with LLVM's assembly language as identifiers (registers) cannot be reassigned. Instead the stack activity of instructions must be traced,

```
LDC 0    ; load constant 0
LDL 0    ; load workspace location 0
LDC 64   ; load constant 64
CSUB0    ; assert stack 1 < stack 0, and pop stack
LDLP 3   ; load a pointer to workspace location 3
BSUB     ; subscript stack 0 by stack 1
SB       ; store byte in stack 1 to pointer stack 0
```

Figure 54: Example ETC which stores a 0 byte to an array. The base of the array is workspace location 3, and offset to be written is stored in workspace location 0.

```
LDC 0    ; () => (reg_1)                STACK = <reg_1>
LDL 0    ; () => (reg_2)                STACK = <reg_2, reg_1>
LDC 64   ; () => (reg_3)                STACK = <reg_3, reg_2, reg_1>
CSUB0    ; (reg_3, reg_2) => (reg_2)    STACK = <reg_2, reg_1>
LDLP 3   ; () => (reg_4)                STACK = <reg_4, reg_2, reg_1>
BSUB     ; (reg_4, reg_2) => (reg_5)    STACK = <reg_5, reg_1>
SB       ; (reg_5, reg_1) => ()         STACK = <>
```

Figure 55: Tracing the stack utilisation of the ETC in figure 54, generating a register for each unique operand.

creating a new identifier for each operand pushed and associating it with each subsequent pop or reference of that operand. This is possible as all ETC instructions consume and produce constant numbers of operands.

The process of tracing operands demonstrates one important property of SSA, its obviation of data dependencies between instructions. Figures 54, 55 and 56 show respectively: a sample ETC fragment, its traced form and a data flow graph derived from the trace. Each generated identifier is a node in the data flow graph connected to nodes for its producer and consumer nodes. From the example it can be seen that only the `SB` instruction depends on the first `LDC`, therefore it can be reordered to any point before the `SB`, or in fact constant folded. This direct mapping to the data flow graph representation, is what makes SSA form desirable for pass-based optimisation.

The Transputer has a two stacks, the standard operand stack and the floating point operand stack. The tracing process is applied to both. A data structure in the translator provides the number of input and output operands for each instruction. Additionally,

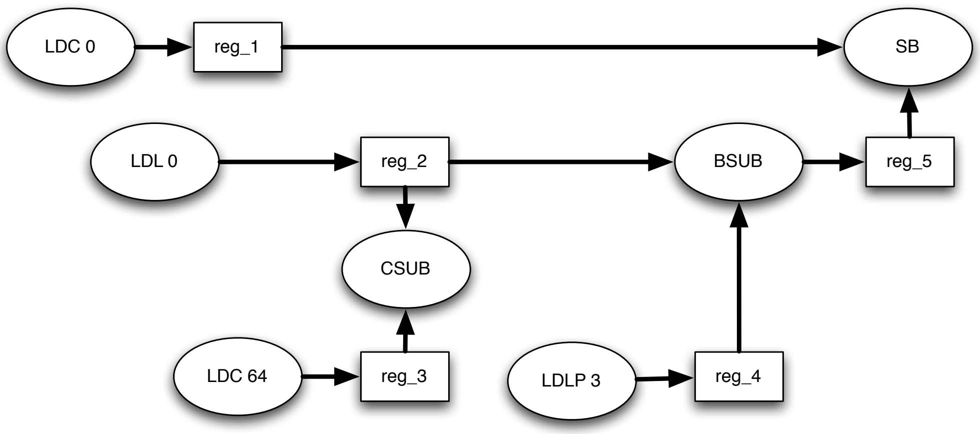


Figure 56: Data flow graph generated from the trace in figure 55.

modifications to the workspace register are traced and it is redefined as required.

Registers from the operand stack are typed as 32-bit integers (`i32`), and operands on the floating point stack as 64-bit double precision floating point numbers (`double`). The workspace pointer is an integer pointer (`i32*`). When an operand is used as a memory address it is cast to the appropriate pointer type. In theory, these casts may hinder certain kinds of optimisations, but in practice no difference in behaviour is observed.

4.3.2 Process Representation

While the programmer's view of occam is one of all processes executing in parallel, this model is in practice simulated by one or more threads of execution moving through the concurrent processes. The execution flow may leave processes at defined instructions, reentering at the next instruction. The state of the operand stack after these instructions is undefined. Instructions which deschedule the process, such as channel communication or barrier synchronisation, are implemented as calls to the runtime kernel (CCSP) (see chapter 3). In a low-level machine code generator such as tranx86, the generator is aware of all registers in use and ensures that their state is not assumed constant across a kernel call. Take the example in figure 57; there is a risk the code generator may choose to remove the second load of workspace offset 1, and reuse the register from

```
; load workspace offset 1
%reg_1 = load i32* (getelementptr i32* %wptr_1, i32 1)
; add 1
%reg_2 = add i32 %reg_1, 1
; store result to workspace offset 2
store i32 %reg_2, (getelementptr i32* %wptr_1, i32 2)

; load workspace offset 3
%reg_3 = load i32* (getelementptr i32* %wptr_1, i32 3)
; synchronise barrier
call void kernel_barrier_sync (%reg_3)

; load workspace offset 1
%reg_4 = load i32* (getelementptr i32* %wptr_1, i32 1)
; add 2
%reg_5 = add i32 %reg_4, 2
; store result to workspace offset 2
store i32 %reg_5, (getelementptr i32* %wptr_1, i32 2)
```

Figure 57: LLVM code example which illustrates the dangers of optimisation across kernel calls.

the first load. However the value of this register may have been changed by another process which is scheduled by the kernel before execution returns to the process in the example.

While the system ABI specifies which registers should be preserved by the callee if modified, the kernel does not know which registers will be used by other processes it schedules. If the kernel is to preserve the registers then it must save all volatile registers when switching processes. This requires space to be allocated for each process's registers, something the occam compiler does not do as the instruction it generated was clearly specified as to undefine the operand stack. More importantly, the code to store a process's registers must be rewritten in the system assembly language for each platform to be supported. Given a goal of minimal maintenance portability this is not acceptable.

The solution is to break down monolithic processes into sequences of uninterruptable functions which pass continuations [214]. Control flow is then restricted such that it may only leave or enter a process at the junctures between its component functions.

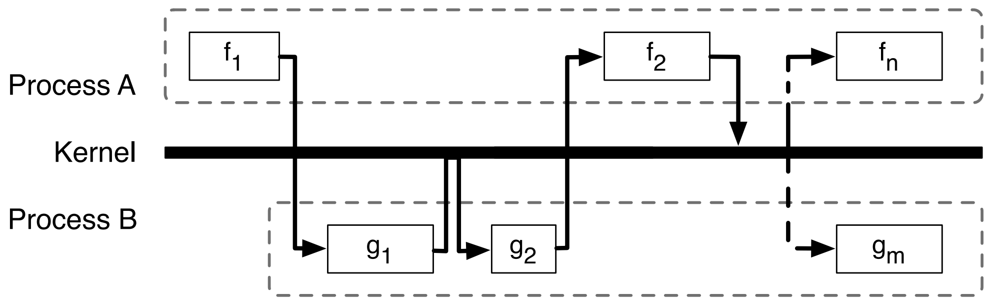


Figure 58: Execution of the component functions of processes A and B is interleaved by the runtime kernel.

The functions of the process are then mapped directly to LLVM function definitions, which gives LLVM an identical view of the control flow to that of the internal representation. LLVM's code generation backends will then serialise state at points where control flow may leave the process. Figure 58 gives a graphical representation of this process, as the runtime kernel interleaves the functions f_1 to f_n of process A with g_1 to g_m of process B.

In practice the continuation is the workspace (`Wptr`), with the address of the next function to execute stored at `Wptr[Iptr]` (see 3.7). This is very similar to the Transputer's mechanism for managing blocked processes, except the stored address is a function and thus the dispatch mechanism is not a jump, but a call. Thus the dispatch of a continuation (`Wptr`) is the tail call: `Wptr[Link] (Wptr).`

The dispatch of continuations is handled by generated LLVM assembly. Kernel calls return the next continuation as selected by the scheduler, which is then dispatched by the caller. This removes the need for system specific assembly instructions in the kernel to modify execution flow, and thus greatly simplifies the kernel implementation. The runtime kernel can then be implemented as a standard system library as its ABI is now consistent with the rest of the system. This contrasts with the compiler toolchain as defined in section 3.5. The call table is entirely removed; this enhances branch prediction as an indirect branch can be replaced with a direct one.

```
; Component function of process "kroc.screen.process"
define private fastcc void @O_kroc_screen_process_L0.3_0
                           (i8* %sched, i32* %wptr_1) {
  ; ... code omitted ...

  ; Build continuation
  ; tmp_6 = pointer to workspace offset -1
  %tmp_6 = getelementptr i32* %wptr_1, i32 -1
  ; tmp_7 = pointer to continuation function as byte pointer
  %tmp_7 = bitcast void (i8*, i32*)* @O_kroc_screen_process_L0.3_1 to i8*
  ; tmp_8 = tmp_7 cast to an 32-bit integer
  %tmp_8 = ptrtoint i8* %tmp_7 to i32
  ; store tmp_8 (continuation function pointer) to workspace offset -1
  store i32 %tmp_8, i32* %tmp_6

  ; Make kernel call
  ; The call parameters are reg_8, reg_7 and reg_6
  ; The next continuation is return by the call as  tmp_9
  %tmp_9 = call i32* @kernel_Y_in
                     (i8* %sched, i32* %wptr_1,
                      i32 %reg_8, i32 %reg_7, i32 %reg_6)

  ; Dispatch the next continuation
  ; tmp_10 = pointer to continuation offset -1
  %tmp_10 = getelementptr i32* %tmp_9, i32 -1
  ; tmp_12 = pointer to continuation function cast as 32-bit integer
  %tmp_12 = load i32* %tmp_10
  ; tmp_11 = pointer to continuation function
  %tmp_11 = inttoptr i32 %tmp_12 to void (i8*, i32*)*
  ; tail call tmp_11 passing the continuation (tmp_9) as its parameter
  tail call fastcc void %tmp_11 (i8* %sched, i32* %tmp_9) noreturn
  ret void
}

; Next function in the process "kroc.screen.process"
define private fastcc void @O_kroc_screen_process_L0.3_1
                           (i8* %sched, i32* %wptr_1) {
        ; ... code omitted ...
}
```

Figure 59: LLVM code example is actual output from the translation tool showing a kernel call for channel input. This demonstrates continuation formation and dispatch.

Figure 59 shows the full code listing of a kernel call generated by translation. Two component functions of a process `kroc.screen.process` are shown (see section 4.3.5 for more details on function naming). The first constructs a continuation to the second, then makes a kernel call for channel input and dispatches the returned continuation.

4.3.3 Calling Conventions

When calling a process as a subroutine, the present process function is split and a tail call made to the callee passing a continuation to the newly created function as the return address. This process is essentially the same as the Transputer instructions `CALL` and `RET`. There are are however some special cases which must be addressed.

The occam language has both processes (`PROC`) and functions (`FUNCTIONS`). Processes may modify their writable (non-`VAL`) parameters, interact with their environment through channels and synchronisation primitives, and go parallel creating concurrent subprocesses. Functions on the other hand may not modify their parameters or perform any potentially blocking operations or go parallel, but may return values (processes do not return values).

While it is possible to implement occam's pure functions in LLVM using the normal call stack, the translation presented here does not address this. Instead function calls are treated as process calls. Function returns are then handled by rewriting the return values into parameters to the continuation function.

The main obstacle to supporting pure functions is that the *occ21* compiler lowers functions to processes; this obscures functions in the resulting ETC output. It also allows some kernel operations (e.g. memory allocation) within functions. Hence to provide pure function support, the translation tool must reconstruct functions from processes, verify their purity, and have separate code generation paths for process and functions. Such engineering is excessive for unknown gains, although as the LLVM optimiser is likely to provide more effective inlining and fusion of pure functions it is an area for further work. In particular, the purity verification stage in such a translator should also be able to lift processes to functions, further improving code generation.

Another area affected by LLVM translation is the Foreign Function Interface (FFI). The FFI allows occam programs to call functions implemented in other languages, such as C [265, 96]. This facility is used to access the system libraries for file input and output, networking and graphics. The KRoC code generator (*tranx86*) must generate not only hardware specific assembly, but structure the call to conform to the operating system specific ABI. LLVM greatly simplifies the FFI call process as it abstracts away any ABI specific logic. Hence foreign functions are implemented as standard LLVM calls in the translator.

4.3.4 Branching and Labels

The Transputer instruction set has a relatively small number of control flow instructions:

- `CALL` call subroutine (and a general call variant - `GCALL`),
- `CJ` conditional jump,
- `J` unconditional jump,
- `LEND` loop end (form of `CJ` which uses a counting block in memory),
- `RET` return from subroutine.

Sections 4.3.2 and 4.3.3 addressed the `CALL` and `RET` related elements of translation. This section addresses the remaining branching instructions.

The interesting aspect of the branching instructions `J` and `CJ` are their impact on the operand stack. An unconditional jump undefines the operand stack; this allows a process to be descheduled on certain jumps, which provided a preemption mechanism for long running processes on the Transputer. The conditional jump instruction branches if the first element of the operand stack is zero; in doing so it preserves the stack. If it does not branch then it instead pops the first element of the operand stack.

As part of operand tracing during the conversion to SSA-form (4.3.1), each label encountered within a process is tagged with the present stack operands. For the purposes of tracing, unconditional jumps undefine the stack and conditional jumps consume the entire stack outputting $stackdepth - 1$ new operands. Having traced the stack the inputs of each label are compared with the inferred inputs from branch instructions which reference it and behaviour is adjusted as required These adjustments can occur, for example, when the target of a conditional jump does not require the entire operand stack. While the compiler outputs additional stack depth information this is not always sufficient, hence the introduction of an additional verification stage.

The SSA syntax of LLVM's assembly language adds some complication to branching code. When a label is the target of more than one branching instruction, ϕ nodes (phi nodes) must be introduced for each identifier which is dependent on the control flow. Figure 60 illustrates the use of ϕ nodes in a contrived code snippet generating $1/n$, where the result is 1 when $n = 0$. The ϕ node selects a value for `%fraction` from the appropriate label's namespace, acting as a merge of values in the data flow graph. During translation the operand stack information generated for each label is used to build appropriate ϕ nodes for labels which are branch targets. Unconditional branch instructions are then added to connect these labels together, as LLVM's control flow does not automatically transition between labels.

Transputer bytecode is by design position independent; the arguments passed to start process instructions, loop ends and jumps are offsets from the present instruction. To support these offsets the occam compiler specifies the instruction arguments as label differences, i.e. $L_t - L_i$ where L_t is the argument's target label and L_i is a label placed before the instruction consuming the jump offset. While these differences can be reverted to absolute label reference by removing the subtraction of L_i, LLVM assembly does not permit the derivation of label addresses.

The inability to derive the address of a label prevents the passing of labels as arguments to kernel calls such as start process (STARTP). This is overcome by lifting labels, for which the address is required, to function definitions. This is achieved by splitting

```
  ; Compare n to 0.0
  %is_zero = fcmp oeq double %n, 0.0
  ; Branch to the correct label
  ;  zero if is_zero = 1, otherwise not_zero
  br i1 %is_zero, label %zero, label %not_zero

zero:
  ; Unconditionally branch to continue label
  br label %continue

not_zero:
  ; Divide 1 by n
  %nz_fraction = fdiv double 1.0, %n
  ; Unconditionally branch to continue label
  br label %continue

continue:
  ; fraction depends on the source label:
  ;   1.0 if the source is zero
  ;   nz_fraction if the source is not_zero
  %fraction = phi double [ 1.0, %zero, %nz_fraction, %not_zero ]
```

Figure 60: Example LLVM code showing the use of a phi node to select the value of the *fraction* identifier.

the process in the same way as is done for kernel calls (4.3.2). Adjacent labels are then connected by tail calls with the operand stack passed as parameters. There is no need to build continuations for these calls as control flow will not leave the process. Additionally, ϕ nodes are not required as the passing of the operand stack as parameters provides the required renaming.

As an aside, early verions of the translation process lifted all labels to function definition to avoid the complexity of generating ϕ nodes, and avoid tracking the use of labels as arguments. While it appeared that LLVM's optimiser was able to fuse many of these processes back together, a layer of control flow was being obscured. In particular this created output which was often hard to debug. Hence, it is desirable to only lift labels when required.

4.3.5 Symbol Naming

While LLVM allows a wide range of characters in symbol names, the generation of symbol names for processes is consistent with that used in *tranx86* [47]. Characters not valid for a C function are converted to underscores, and a `O_` prefix added. This allows ANSI C code to manipulate occam process symbols by name.

Only processes marked as global by the compiler are exported, and internally generated symbols are marked as `private` and tagged with the label name to prevent internal collisions. Declaration `declare` statements are added to the beginning of the assembly output for all processes referenced within the body. These declarations may include processes not defined in the assembly output; however, these will have been validated by the compiler as existing in another ETC source. The resulting output can then be compiled to LLVM bytecode or system assembly and the symbols resolved by the LLVM linker or the system linker as appropriate.

4.3.6 Arithmetic Overflow

An interesting feature of the occam language is that its standard arithmetic operations check for overflow and trigger an error when it is detected. In the ANSI C TVM emulating these arithmetic instructions requires a number of additional logic steps and calculations [146]. This is inefficient on CPU architectures which provide flags for detecting overflow. The LLVM assembly language does not provide access to the CPU flags, but instead provides intrinsics for addition, subtraction and multiplication with overflow detection. These intrinsics (`@llvm.sadd.with.overflow`, `@llvm.ssub.with.overflow` and `@llvm.smul.with.overflow`) can be used to efficiently implement the instructions `ADD`, `SUB` and `MUL`.

4.3.7 Floating Point

The occam language supports a wide range of IEEE floating-point arithmetic and provides the ability to set the rounding mode in number space conversions. While an emulation library exists for this arithmetic, a more efficient hardware implementation is present in later Transputers and this translator. However, LLVM lacks support for setting the rounding mode of the floating point unit. The LLVM assembly language specification defines all the relevant instructions to truncate their results. While not ideal, this truncation can be used by adding or subtracting 0.5 before converting a value in order to simulate nearest rounding. While *plus* and *minus* rounding modes are defined by convention the *occ21* compiler never generates instructions for these so do not need to be supported.

It can be observed that the *occ21* compiler only ever generates a rounding mode change instruction directly prior to a conversion instruction. Thus instead of generating LLVM code for the mode change instruction the translation tags the proceeding instruction with the new mode. Hence mode changes become static at the point of translation and can be optimised by LLVM.

4.4 Analysis

In this section contains benchmark results comparing the output of the existing *tranx86* ETC converter to the output of the new translation tool passed through LLVM's optimiser (*opt*) and native code generator (*llc*). These benchmarks were performed using source code as-is from the KRoC subversion repository revision *6002* [1], with the exception of the *mandelbrot* benchmark from which the frame rate limiter was removed.

LLVM version 2.7 was used with the following optimisations:

- `constmerge` and `constprop` to merge and simplify constants
- `mergefunc` merge functions
- `die` dead instruction elimination
- `dce` dead code elimination

[1] `http://projects.cs.kent.ac.uk/projects/kroc/trac/log/kroc/trunk?rev=6002`

- `scalar-evolution` scalar evolution analysis
- `lcssa` loop dependence analysis
- `memcpyopt` memory copy optimisation
- `mem2reg` promote memory to registers
- `ipconstprop` inter-procedural constant propagation
- `dse` dead store elimination
- `globalopt` global variable optimiser
- `break-crit-edges` break critical edges
- `loop-deletion` delete dead loops
- `loopsimplify` canonicalize natural loops
- `jump-threading` jump threading
- `libcall-aa` libcall alias analysis
- `simplify-libcalls` simplify well-known library calls
- `simplifycfg` simplify control flow graph

Table 5 shows the wall-clock execution times of the various benchmarks. All our benchmarks were performed on an eight core Intel Xeon workstation composed of two E5320 quad-core processors running at 1.86GHz. Pairs of cores share 4MiB of L2 cache, giving a total of 16MiB L2 cache across eight cores.

4.4.1 agents

The *agents* benchmark was developed to compare the performance of the CCSP runtime to that of other language runtimes (see section 3.15.4). The amount of computation increases greatly with the density of agents and hence two variants were run for comparison: one with 32 initial agents per grid tile on an eight by eight grid, giving 2048 agents,

Table 5: Benchmark execution times, comparing tranx86 and LLVM based compilations. Confidence interval of 95%.

Benchmark	tranx86 (s) CI 95%	LLVM (s) CI 95%	Difference (tranx86 $\rightarrow$ LLVM)
agents 8 32	17.50 - 17.55	15.97 - 16.00	-9%
agents 8 64	62.49 - 62.56	56.38 - 56.47	-10%
fannkuch	1.272 - 1.272	1.315 - 1.316	+3%
fasta	6.236 - 6.241	6.646 - 6.712	+6%
mandelbrot	17.54 - 17.54	7.120 - 7.381	-58%
ring 250000	2.994 - 3.020	3.945 - 3.981	+31%
spectralnorm	22.96 - 22.97	13.56 - 14.23	+38%

and the other with double the density at 4096 agents on the same size grid. A marginal performance improvement can be seen in the LLVM version of this benchmark. This can be attributed to LLVM's aggressive optimisation of the computation loops.

4.4.2 fannkuch

The *fannkuch* benchmark is based on a version from *The Computer Language Benchmarks Game* [13, 33]. The source code involves a large numbers of reads and writes to relatively small arrays of integers. A small decrease in performance can be seen in the LLVM version of this benchmark. This may be the result of tranx86 generating a more efficient instruction sequence for array bounds checking.

4.4.3 fasta

The *fasta* benchmark is also taken from *The Computer Language Benchmarks Game*. A set of random DNA sequences is generated and output, this involves array accesses and floating-point arithmetic. Again, like *fannkuch*, a negligible decrease in performance can be seen and again this can be attributed to array bounds checks.

4.4.4 mandelbrot

As used in section 3.15.2, the occam-pi implementation of the mandelbrot set generator in the `ttygames` source directory was modified to remove the frame rate limiter and used this as a benchmark. The implementation farms lines of the mandelbrot set image to 32 worker processes for generation, and buffers allow up to eight frames to be concurrently calculated. The complex number calculations for the mandelbrot set involve large numbers of floating point operations, and this benchmark demonstrates a vast improvement in LLVM's floating-point code generator over tranx86. FPU instructions are generated by tranx86, whereas LLVM generates SSE instructions; the latter appear to be more efficient on modern x86 processors. Additionally, by tracking the rounding mode at the source level (4.3.7) the need to generate FPU mode change instructions is removed. These instructions may disrupt FPU pipelining in tranx86 generated code.

4.4.5 ring

Another CCSP comparison benchmark (see 3.15.3). This benchmark sets up a ring of 256 processes. Ring processes receive a token, increment it, and then forward it on to the next ring node. The benchmarks here time 250000 iterations of the ring, giving 64000000 independent communications. This allows calculation of the communication times (based on upper bound) of the tranx86 and LLVM implementation at 47ns and 62ns respectively. The increased communication time can be attributed to the additional instructions required to unwind the stack when returning from kernel calls in the LLVM implementation. The tranx86 version of CCSP does not return from kernel calls (it dispatches the next process internally).

4.4.6 spectralnorm

The final benchmark from *The Computer Language Benchmarks Game*. This benchmark calculates the spectral norm of an infinite matrix. Matrix values are generated using floating-point arithmetic by a function which is called from a set of nested loops. The

Table 6: Binary text section sizes, comparing tranx86 and LLVM based compilations.

Benchmark	tranx86 (bytes)	LLVM (bytes)	Difference (tranx86 $\rightarrow$ LLVM)
agents	16410	36715	+124%
fannkuch	3702	5522	+49%
fasta	5134	10494	+104%
mandelbrot	6098	12865	+111%
ring	3453	6716	+94%
spectralnorm	4065	6318	+55%

significant performance reduction with LLVM can be attributed to its excessive inlining causing cache overloads negating the impact of potentially more efficient floating-point code generation. This suggest some revision of optimisations is required.

4.4.7 Code Size

Table 6 shows the size of the text section of the benchmark binaries. It can be seen that the LLVM output is typically twice the size of the equivalent tranx86 output. It is surprising that this increase in binary size does not adversely affect performance; increased binary size potentially reduces cache space available for program data. As an experiment the `—code—model=small` option was passed to LLVM's native code generator; however, this made no difference to binary size. Some of the increase in binary size may be attributed to the continuation dispatch code which is inlined within the resulting binary, rather than as part of the runtime kernel as with tranx86. The *fannkuch* and *spectralnorm* benchmarks make almost no kernel calls, therefore contain very few continuation dispatches, and accordingly show the least growth. Another possibility is LLVM aggressively aligning instructions to increase performance. The increase in binary size is of concern when targeting memory constrained embedded devices.

4.4.8 Summary

Figure 61 shows the combined results for speed up and binary size growth. Code size growth is very consistent. Performance is boosted significantly when floating point

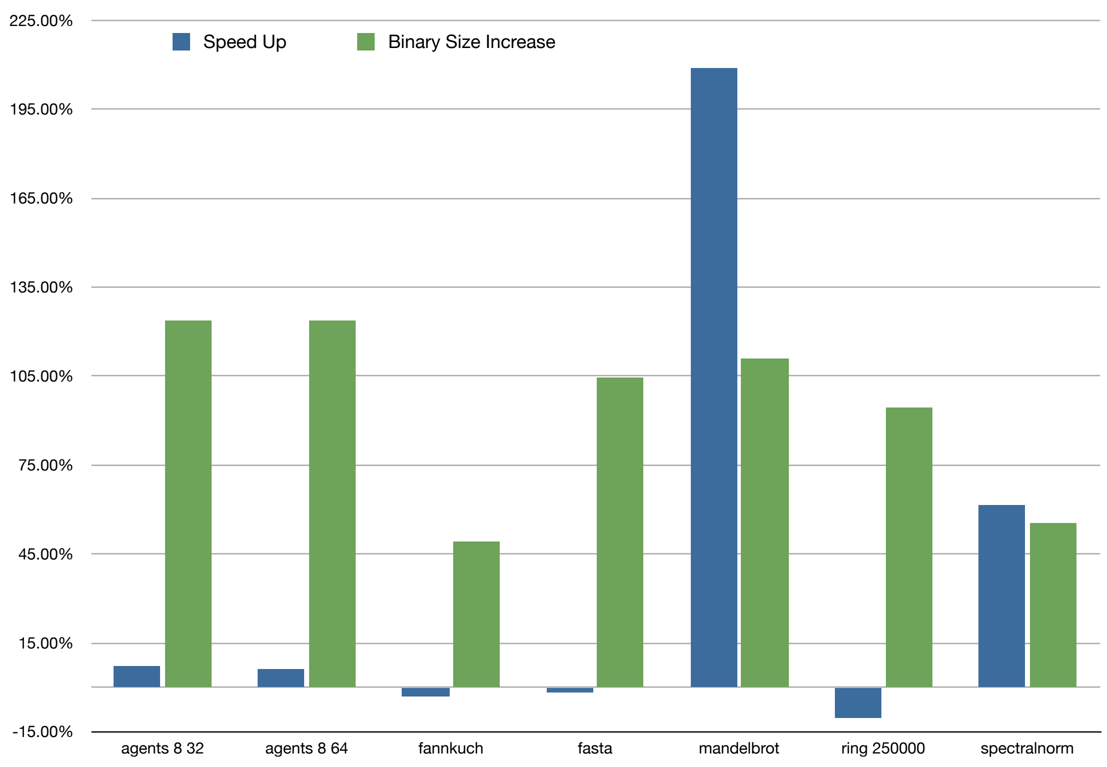


Figure 61: Overview of performance speed up and binary size growth with LLVM compilation.

computation is involved. Communication time is marginally increased, this hurts the communication only benchmark *ring*, but not catastrophically. However, the agents benchmark which has a high degree of process communication and no floating point computation still shows a speed up suggesting overall optimisation is good. The original *occoids* program uses floating point so is likely to see further performance increases.

4.5 Conclusions

This work has demonstrated the feasibility of translating the ETC output of the present occam-pi compiler into the LLVM project's assembly language. With associated changes to the runtime kernel this work provides a means of compiling occam-pi code for platforms other than Intel x86. This can also be seen as a stepping stone on the path to direct compilation of occam-pi using LLVM assembly as part of a new compiler, such as *Tock* [14]. In particular, viable representations of processes and a kernel calling convention have been established, both fundamental details of any compiled representation of occam-pi and other process-oriented languages.

The performance of LLVM translations compares favourably with benchmarks of the modified KRoC runtime using tranx86 (4.4). While the LLVM kernel call mechanism is approximately 10% slower, loop unrolling enhancements and dramatically improved float-point performance offset this overhead. Typical applications are a mix of communication and computation, which should help preserve this balance. The *occ21* compiler's memory bound model of compilation presents an underlying performance bottleneck to translation based optimisations. This is a legacy of the Transputer's limited number of stack registers.

Aside from the portability aspects of this work, access to an LLVM representation of occam-pi programs opens the door to exploring concurrency specific optimisations within an established optimisation framework. Interesting optimisations, such as fusing parallel processes using vector instructions and removing channel communications in linear pipelines (see section 6.3), could be implemented as LLVM passes. LLVM's bytecode has also been used for various forms of static verification, a similar approach many be able to verify aspects of a compiled occam-pi program such as the safety of its access to mobile data. LLVM's assembly language may benefit from a representation of concurrency, particularly for providing information to concurrency related optimisations. Recent research has begun targeting this area [97].

Chapter 5

Introspection

This chapter looks at how a runtime for a process-oriented language, in this case occam-pi, can support low-level introspection of execution as a means for building debugging and visualisation tools. The goal of this work is to investigate how runtime support can be used to facilitate introspective debugging where concurrent components observe and interact with other concurrent components. Work in this chapter has been previously published as [221].

5.1 Motivation

While a program may be correct by design, as long as there remains a gap between that design and its implementation then an opportunity exists for errors to be introduced. In the ideal world our implementations would be automatically verified against higher-level specifications of our designs, but despite much work in the area, such checking is not yet supported for all cases. Hence support for finding and fixing errors is essential to the rapid development of large-scale and complex applications. Debugging is the process of locating, analyzing, and correcting suspected errors [178].

The Kent Retargetable occam-pi Compiler (KRoC) supports basic post-mortem debugging [266], which gives the developer access to the last executed line and process call at the time a fatal error such as integer overflow or array bounds violation is detected. This provides the developer with a starting point for debugging and assists with the correction of many trivial errors; however, it does not elucidate the, often complex, interactions which lead up to application failure, nor does it provide any assistance for non-fatal errors. The work presented in this chapter is an extension of earlier work and is aimed at providing the developer with uniform and accessible tracing and replay facilities for occam-pi applications.

Without any explicit debugging support developers must implement their own tracing support. This typically takes the form of a shared messaging channel along which component processes emit their status. The programmer writes "print" commands into critical sections of the application and then views the trace output. Concurrency, however, creates many issues for this debugging approach. Buffering on the messaging channel or the presence of multiple threads of execution will cause the trace output to deviate from the actual execution flow of the application. This hinders the detection of errors in the interaction between processes. In turn, the introduction of the messaging channel itself, a new shared resource on which component processes synchronise, subtly changes the scheduling behaviour of the program obscuring race condition related faults. This is often call the *probe effect* and it is further defined in section 5.6.1.

This chapter briefly reviews previous work in the field of parallel application debugging, and more specifically the debugging of occam programs (section 5.2). Following on a new method for run-time monitoring of occam-pi applications is introduced. This is based on the Transterpreter [147] virtual machine interpreter. Section 5.4 describes how virtual machine instances can interact with the interpreter to intercede on each other's execution. This allows an occam-pi process to mediate the execution of another occam-pi program, a model which bears similarities to that applied in debugging embedded systems with a JTAG connection [15]. A new extensible bytecode format which provides access to debugging information is detailed in section 5.5. In section 5.6 additional run-time debugging support is described; this has been added to the Transterpreter virtual machine. Then in section 5.7 virtual machine introspection is applied to the task of tracing occam-pi programs.

5.2 Related Work

Viewing the state of an executing program is one method of understanding its behaviour and debugging program code. Past work in the area of debugging occam has concentrated heavily on networks of Transputers, as these were the primary target for large scale occam application development. As such, tracing and debugging of program execution across networks of processors increased the complexity of past solutions. In addition to this observation, past solutions can be divided into those that traced and interacted with running programs and those that acted on stored post-mortem traces.

Stepney's GRAIL [239] extracted run-time data from a running network of Transputers, instrumenting the target occam program at compile-time with a modified version of the INMOS compiler and extracting the data over a custom bus external to the Transputer network [240], so as not to interrupt the communications occurring on the communications links. Maintaining the original timings and communication semantics as a program not being debugged is critical.

Cai and Turner identified in [63] the danger of added debugging hook processes altering the timing of a parallel program and propose a model under which a *monitor* has execution control over all processes and a logical clock is maintained for the execution of each process. The logical clock provides a way to measure and balance the interaction added to the network by the monitor hooks, and is similar to the approach described in section 5.6.1. The paper also identifies the problem of maintaining timings for real-time systems whilst monitoring the system, suggesting the use of dummy test-sources or buffering of real-time data.

Debugging methodologies that require annotations or changes to the code have the potential to introduce additional complexity, changing the run-time dynamics of the program. May's Panorama [175], designed for the debugging of message-passing systems, identifies an approach for post-mortem debugging which records a log of communications. Replaying these communications offers a look at the particular execution of the program whilst introducing only a minimal overhead at run-time. The Panorama system had the downside of requiring all communications to be modified to use a different call structure and the software recompiled against a custom library. This need for program modification, instrumentation or annotation is common to many approaches that do not have the benefit of an interpreted run-time environment.

The INMOS Transputer Debugging System (TDS) [197] allowed the user to inspect the occam source and display run-time values of variables. Programs for use with TDS were compiled in a debug mode which embedded extra information: workspace offsets, types and values of constants, protocol definitions, and workspace requirements for procedures and functions. The TDS provides access to running processes in the network by "jumping through" channels to the process on the other end of the channel. Deadlock detection requires source modification, as a deadlocked process cannot be jumped to using the channel jump system. Processes suspected of causing deadlock must be modified to include a secondary process and channel which will not deadlock and allows a jump into the process for state inspection.

Finally, Zhang and Marwaha's Visputer [267] provided a highly developed visual tool for editing and analysing occam process networks. An editing suite allowed the assisted building of programs using toolkit symbols. A network editor facilitated the configuration of networks of Transputer processors. Pre-processing of source files inserted instrumentation library calls to support post-mortem debugging and performance profiling. A static analyser also predicted performance and detected deadlocks. Importantly Zhang and Marwaha pointed out that occam "has a language structure that is highly suitable for graphical representation". The work presented in this chapter is intended to provide a means of extracting the information required to exploit this feature of occam.

5.3 The Transterpreter Virtual Machine

The Transterpreter, or TVM (Transterpreter Virtual Machine), is a virtual machine interpreter for running occam-pi applications compiled to a modified form of Transputer bytecode [147, 146]. Written in platform independent ANSI C, the TVM emulates a hybrid T8 Transputer processor. Most T8 instructions are supported, with additional instructions added to support dynamics and mobility added by occam-pi. This provides an alternate compilation and execution path to KRoC by emulating the Transputer processor itself rather than converting Transputer instructions to native machine instructions.

The Transputer was a three-place stack machine, and executed a bytecode where the most common instructions and their immediates required only a single byte to represent. Large instructions were composed from sequences of smaller instructions prior to execution. Hence the Transputer bytecode is compact, and simple to interpret. A modified version of the INMOS compiler with occam-pi support is used to generate *extended transputer code* (ETC) a process shared with KRoC. A separate linker then resolves symbolic references and expands certain operations, converting ETC into bytecode for the TVM.

The TVM has a very small memory footprint making it suitable for use on small embedded systems, such as the LEGO Mindstorms RCX [235] and Surveyor SRV-1 [12] mobile robotics platform. The work described here focuses on execution on desktop class machines; however, the portability of the bytecode representation used by the TVM allows emulation of embedded systems on a desktop machine. The techniques detailed here can be used in such a manner to debug embedded application code via emulation when the target platform has insufficient resources to support debugging in-place.

5.4 Introspection

To support this work the TVM was transitioned from a static library design to a fully re-entrant implementation. This shift allows the execution of multiple virtual Transputers concurrently, or at least with the appearance of concurrency. The virtual Transputers can communicate with each other, or more specifically processes in one TVM instance can communicate with processes in another, efficiently and transparently. Being software defined, these Transputers are not restricted in the number of communications links they support, and hence one virtual link is provided per-shared channel, freeing the programmer from multiplexing a fixed number of links. Mobile data [50] and mobile channels [51] are also supported, allowing the construction of arbitrary process networks which in turn span multiple virtual machines.

Each TVM instance has its own registers and run-queues. Instances can be scheduled cooperatively (only switched when they relinquish control), or pre-emptively (time-sliced). The specific algorithm used to schedule between TVM instances is defined by the virtual machine wrapper. For most purposes a round-robin algorithm is used (the same as occam processes); however, priority is also supported.

On a small robotics platform, the Surveyor Corporation SRV-1 [12], multiple TVM instances were used to execute a cooperatively scheduled firmware and a time-sliced user application. The user application is loaded at run-time, and may be swapped out, terminate or even crash, without interfering with the firmware. The firmware executes as required, mediating access to the hardware on behalf of the user application.

The virtual Transputers, TVM instances, allow for the isolation and encapsulation required to start allowing one occam-pi program to mediate and intercede on the execution of another. This concept, of multiple concurrent virtual machine interpreters, underpins the debugging framework presented in later sections.

```
DATA TYPE VM.STATE
  PACKED RECORD
    INT         state:
    [3]INT      stack:
    [4]BYTE     type:
    INT         oreg:
    ADDR        wptr:
    IPTR        iptr:
    INT         icount:
    INT         eflags:
:
```

Figure 62: Virtual machine state data record.

5.4.1 Interface

The application running in each virtual machine instance can access the virtual machine runtime via a special `PLACED` channel. Channel read requests to the `PLACED` channel return a *channel bundle* (channel type [51]) which provides request and response channels for the manipulation of the current virtual machine instances and the creation of new sub-instances. For each sub-instance created a further channel bundle is returned that can be used to control the instruction-by-instruction execution of the sub-instance.

The virtual machine interface channel also provides access to the interpreter's bytecode decoder. Using this interface, a virtual machine can load bytecode into new sub-instances, and access additional information stored in the bytecode. Details of the bytecode format and decoder interface can be found in section 5.5.

Having decoded bytecode a VM instance is created and associated with it. Once *top level process* parameters are supplied, and the instance started, the control interface can be used to mediate its execution in a number of ways. The following subsections detail requests which can be used to control execution. The occam-pi virtual machine state data structure is shown in figure 62. The protocol definitions for manipulating the virtual machine are shown in figures 63 and figure 64 for request and response respectively.

```
PROTOCOL P.VM.CTL.RQ
  CASE
    run            = 0 ; INT
       -- run until for N instructions or until breakpoint
    step           = 1
       -- step traced instruction
    dispatch       = 2 ; INT; INT
       -- dispatch an arbitrary instruction, with argument
    set.bp         = 3 ; IPTR
      -- set break point
    clear.bp       = 4 ; IPTR
      -- clear break point
    get.clock      = 5
      -- get clock details
    set.clock      = 6 ; INT; INT
      -- set clock type and frequency
    trace          = 7 ; INT; BOOL
      -- enable/disable trace type (instruction)
    get.state      = 8
      -- get VM state
    set.state      = 9 ; VM.STATE
      -- set VM state
    read.word      = 10; ADDR
      -- read word at address
    read.byte      = 11; ADDR
      -- read byte at address
    read.int16     = 12; ADDR
      -- read int16 at address
    read.type      = 13; ADDR
      -- read type of memory at address
    return.param   = 14; INT
      -- release parameter N
    set.param.chan = 15; INT; MOBILE.CHAN
      -- set parameter N to channel
:
```

Figure 63: Virtual machine control request protocol.

```
PROTOCOL P.VM.CTL.RE
  CASE
    decoded        = 0 ; IPTR; INT; INT
      -- new IPTR, instruction, arg
    dispatched     = 1 ; IPTR; ADDR
      -- new IPTR and WPTR
    bp             = 2 ; IPTR
      -- break pointer IPTR reached
    clock          = 3 ; INT; INT
      -- clock type and frequency
    ok             = 4
    error          = 5 ; INT
    state          = 6 ; VM.STATE
    word           = 7 ; INT
    byte           = 8 ; BYTE
    int16          = 9 ; INT16
    type           = 10; INT
    channel        = 11; MOBILE.CHAN
:
```

Figure 64: Virtual machine control response protocol.

5.4.1.1 run

Execute bytecode for a number of instruction dispatches, or until a breakpoint or error is encountered. With the exception of breakpoints this causes the sub-instance to act as a normal virtual machine instance. This operation is implemented synchronously and is uninterruptable, blocking execution of the parent virtual machine; however, this could be enhanced to permit interleaved execution. Where processing facilities exist the sub-instance could also be executed concurrently on a separate processor.

5.4.1.2 step

Decode and dispatch a single bytecode instruction. Feedback is given when the instruction is decoded and then after it is dispatched, allowing the supervising process to keep track of execution state without querying the entire virtual machine state.

5.4.1.3 dispatch

Execute, *dispatch*, an instruction not contained in the program bytecode. This can be used to alter program flow, for example to execute a *stop process* instruction to pause the running process and scheduling the next. Alternatively this request can be used to inject a completely new stream of instructions into the virtual machine instance.

5.4.1.4 get.state / set.state

Get and set requests for the state provide access to the virtual machine registers, instruction pointer, operand stack and clock. By combining these requests with the *dispatch* request a debugger can save virtual machine state, change processes and later restore state.

5.4.1.5 read / write

Read and write requests for all basic types provide access to the virtual machine's memory. As these can only be executed when the virtual machine is stopped, they have predictable results outside their influence on program behaviour. If virtual memory is in use then address translation occurs transparently.

5.5 Bytecode

To support debugging enhancements a new extensible bytecode format was developed for the TVM. Until the introduction of this new format, a number of different fixed formats were used for the different platforms supported by the TVM. By creating a new format bytecode decoding support code was unified, and the need to rewrite the decoder when the bytecode is extended has been removed.

5.5.1 Encoding

The new encoding presented here is called *TEncode*. TEncode is a simple binary markup language, a modified version of IFF.

TEncode streams operate in either 16-bit or 32-bit mode, which affects the size of integers used. A stream is made up of *elements*. Each element consists of a 4-byte identifier, followed by an integer, then zero or more bytes of data. Elements always begin on integer aligned boundaries, the preceding element is padded with null bytes ("\0") to maintain this. All integers are big-endian encoded, and of consistent size (2-bytes or 4-bytes) throughout a stream. Integers are unsigned unless otherwise stated.

```
Identifier  := 4 * BYTE
Integer     := INT16 / INT32
Data        := { BYTE }
Padding     := { "\0" }
Element     := Identifier, Integer, Data, Padding
```

Figure 65: TEncode identifiers.

Identifiers are made up of four ASCII encoded characters stored in 8-bit bytes, and are case-sensitive. The last byte indicates the type of data held in the element. The following types are defined:

- **B**yte string. Integer field encodes number of bytes in Data field, excluding padding null-bytes.
- **I**nteger. Integer field encodes signed numeric value, no Data bytes follow.
- **L**ist of elements. Integer field encodes number of bytes in Data field which will be a multiple of the integer size, Data field contains nested elements, and may be parsed as a new or sub-Stream.
- **S**tring (UTF8 null-terminated). Integer field encodes number of bytes in Data including null terminator, but not padding.
- **U**nsigned integer. Integer field encodes unsigned numeric value, no Data bytes follow.

With the exception of signed and unsigned integer types, the *Integer* of all elements defines the unpadded size of the *Data* which follows. Decoders may use this relationship to skip unknown element types, therefore this relationship must be preserved when adding new types.

A TEncode stream begins with the special `tenc` or `TEnc` element, the integer of which indicates the number of bytes which follow in the rest of the stream. A lower-case `tenc` indicates that integers are small (16-bit), whereas an upper-case `TEnc` indicates integers are large (32-bits). The `TEnc` element contains all other elements in the stream.

```
ByteString   := 3 * BYTE, "B", Integer, Data [, Padding ]
SignedInt    := 3 * BYTE, "I", Integer
ElementList  := 3 * BYTE, "L", Integer, Stream
UTF8String   := 3 * BYTE, "S", Integer, {<character byte>},
                "\0" [, Padding ]
UnsignedInt  := 3 * BYTE, "U", Integer

Element      := ByteString / SignedInt / ElementList / UTF8String /
                UnsignedInt

Header       := ("tenc", INT16) / ("TEnc", INT32)
Stream       := { Element }
TEncode      := Header, Stream
```

Figure 66: TEncode types.

Decoders ignore elements they do not understand or care about. If multiple elements with the same identifier exist in the same stream and the decoder does not expect multiple instances then the decoder uses the first encountered. When defining a stream, new elements may be freely added to its definition across versions; however, the order of elements must be maintained in order to keep parsing simple.

5.5.2 Structure

The base Transterpreter bytecode is a TEncode stream defined in figure 67.

As previously stated, the `TEnc` element marks the beginning of a TEncode stream.

```
TEnc <stream length>
  tbcL <length>
    endU <endian (0=little, 1=big)>
    ws U <workspace size (words)>
    vs U <vectorspace size (words)>
    padB <length> <padding>
    bc B <length> <bytecode>
```

Figure 67: TEncode header fields.

The stream contains a number of `tbcL` elements, each defines a bytecode chunk. A stream may contain multiple bytecode chunks in order to support alternative compilations of the same code, for example with different endian encodings. The `endU` element specifies the endian type of the bytecode. The `ws U` and `vs U` elements specify the workspace and vectorspace memory requirements in words. The `padB` element ensures there is enough space to in-place decode the stream. Finally, the `bc B` element contains the bytecode which the virtual machine interpreter executes.

Following the mandatory elements a number of optional elements specify additional properties of a chunk of bytecode. By placing these elements after the mandatory elements a stream decoder on a memory constrained embedded system can discard all unnecessary elements, such as debugging information, once the mandatory elements have been received and decoded.

A `tlpL` element defines the arguments for the *top level process* (entry point). The foreign function interface table, and associated external library symbols are provided in a `ffiL` element. A symbol table, defining the offsets, memory requirements and type headers of processes within the bytecode is provided by a `stbL` element. Finally a `dbgL` element specifies debugging information.

The debugging information takes the form shown in figure 68.

A table of source file names is defined by `fn S` elements. A table of integer triples is then specified by the `lndB` element. The integers in each triple correspond to a bytecode offset, the index of the source file (in the source file name table), and the line number in the specified source file. The table is arranged such that bytecode offsets are ascending

```
dbgL <length>
  // File names
  fn L <length>
    fn S <length> <file name>
  // Line Numbering Data
  lndB <length>
    <bytecode offset> <file index> <line number>
    ... further entries ...
```

Figure 68: TEncode debugging section.

and offsets that fall between entries in the table belong to the last entry before the offset. For example if entries exist for offsets 0 and 10, then offset 5 belongs to offset 0.

5.5.3 In-place Decoding

On memory constrained embedded platforms it is important to minimise and preferably remove dynamic memory requirements. For this reason the new bytecode format is designed not to require dynamic memory allocation for decoding. This is achieved by providing for rewriting of the bytecode in memory as it is decoded. The C structure `tbc_t` is placed over the memory of the `tbcL` element of the TEncode stream and the component fields written as their TEncode elements are decoded. The `bytecode` field is a pointer to the memory address of the data of the `bc B` element. `tlp`, `ffi`, `symbols` and `debug` fields are pointers to the in-place decodes of their associated elements. The pointers are `NULL` if the stream does not contain the associated element.

5.5.4 Interface

As with the introspection interface discussed in section 5.4, the bytecode decoder also has a channel interface accessible from within a virtual machine instance. When passing an encoded bytecode array to the virtual machine, a channel bundle is returned which can be used to access the decoded bytecode. The following subsections detail some of the available requests. The occam-pi protocol definition for the channel bundle is shown in figure 70.

```
struct tbc_t {
  unsigned int  endian;
  unsigned int  ws;
  unsigned int  vs;

  unsigned int  bytecode_len;
  BYTE         *bytecode;

  tbc_tlp_t    *tlp;
  tbc_ffi_t    *ffi;
  tbc_sym_t    *symbols;
  tbc_dbg_t    *debug;
};
```

Figure 69: TEncode in-place decoding structure.

5.5.4.1 create.vm

Create a new virtual machine instance based on this bytecode. A control channel bundle is returned for the new instance.

5.5.4.2 get.file

Translate a source file index to its corresponding file name. The file name is returned as a mobile array of bytes.

5.5.4.3 get.line.info

Look up the line numbering information for a bytecode address. The source file index and line number are returned if the information exists.

5.5.4.4 get.symbol / get.symbol.at

Look up a process symbol by name or bytecode address. The symbols bytecode offset, name, type description and memory requirements are returned if the symbol exists.

```
PROTOCOL P.BYTECODE.RQ
  CASE
    create.vm       = 0
    get.symbol      = 1; MOBILE []BYTE
      -- look up symbol name
    get.symbol.at   = 2; IPTR
      -- look up symbol at bytecode offset
    get.file        = 3; INT
      -- translate file number to name
    get.line.info   = 4; IPTR
      -- get file/line number of address
    get.details     = 5
      -- get bytecode details
    get.tlp         = 6
      -- get top-level-process details
:
PROTOCOL P.BYTECODE.RE
  CASE
    vm              = 0; CT.VM.CTL!
    error           = 1; INT
    file            = 2; MOBILE []BYTE
    line.info       = 3; INT; INT       -- file, line
    symbol          = 4; IPTR; MOBILE []BYTE; MOBILE []BYTE; INT; INT
      -- offset, name, definition, ws, vs
    details         = 5; INT; INT; INT  -- ws, vs, length
    tlp             = 6; MOBILE []BYTE; MOBILE []BYTE
:
CHAN TYPE CT.BYTECODE
  MOBILE RECORD
    CHAN P.BYTECODE.RQ request?:
    CHAN P.BYTECODE.RE response!:
:
```

Figure 70: Bytecode channel bundle protocol definitions.

5.5.4.5 get.tlp

Access information on the top-level-process (entry point), if one is defined in the bytecode. If defined, the format mask and symbol name of the process are returned. This information is required to setup the entry point stack.

5.6 Debugging Support

The following section details significant changes and features added to the TVM to further support debugging of occam-pi programs.

5.6.1 The Probe Effect

The *probe effect* is a term used to describe the consequence that observing a parallel or distributed system may influence its behaviour. This is a reminiscent of the Heisenberg uncertainty principle as applied in quantum physics. If the decisions made in non-deterministic elements of a program differ between runs under and not under observation, then the observed behaviour will not be the actual operation behaviour of the program. The impact of this on debugging is that, on observation of a known faulty program, errors do not occur or different errors appear.

To prevent the occurrence of the probe effect we must ensure the non-deterministic behaviour of the program is not altered by observation. In occam-pi there are two sources of programmed non-determinism: alternation constructs and timers. A program free of these constructs is completely deterministic; in fact a program is still deterministic in the presence of simple timer delays [71].

Previous work has shown that by maintaining the sequence of events in a program and recording the decisions made by non-deterministic elements, it is possible to replay parallel programs [162]. In the work presented here no attempt is made to constrain the non-determinism which occurs from external inputs to the program, only to control the internal non-determinism. Given that the implementation of virtual machine constructs such as alternation is unaffected by changes made to monitor program execution, only the non-determinism caused by timers needs to be managed.

The method used for constraining timer based non-determinism is to use a logical time clock, as opposed to a real time clock [63]. This logical clock ticks based on the count of instructions executed. By applying a suitable multiplier the instruction count is scaled to give a suitable representation of time.

The accuracy of the logical clock's representation of time depends on the scaling value used. At program start-up the virtual machine (or firmware components of it), calculates the average execution time of a spread of instructions, and uses this to derive a scaling factor. This scaling factor is then periodically adjusted and offset, such that the clock time matches the actual time required to execute the program so far. If virtual machine execution pauses, for example waiting on the timer queue, then an adjustment need only be stored to the new time when execution resumes.

In order to replay a program's execution only the initial scaling factor and the instruction offset and value of subsequent adjustments need to be stored. If adjustments are stored in three 32-bit integers, then any single adjustment will be 12 bytes in size. Assuming adjustments are made approximately once a second then around 40KiB of storage is required for every hour of execution. As the program is replayed, adjustments are applied at their associated instruction offsets irrespective of the actual execution speed of the program.

5.6.2 Memory Shadowing

Support for shadowing each location of the *workspace* (process stack) with type information of the value it holds was added. This is, in principle, similar to other memory checking and debugging tools such as Valgrind [192].

Type information assists in the debugging of programs in the absence of additional compiler information about the memory layout. The type mapping is maintained at run-time for the virtual machine registers and copied to and from the shadow map as instructions are interpreted. For example if a pointer to a workspace memory location is loaded and then stored to another memory location, then the type shadow will indicate the memory is a workspace pointer. The shadow types presently supported are:

- *Data*: The memory location holds data of unknown type.
- *Workspace*: Pointer to a workspace memory location.
- *Bytecode*: Pointer to a location in bytecode memory, e.g. an instruction or constant pointer.
- *Mobile*: Pointer into mobile memory.
- *Channel*: Memory is being used as a channel.
- *Mobile type*: Mobile type pointer or handle.

By iterating over the shadow of the entire memory, a debugger can build an image of a paused or crashed program. To assist in this, the memory shadow holds flags about whether the memory is utilised or not. Workspace words are marked as in-use when they write to, and marked as free whenever a process releases them via the `ADJW` (*adjust workspace*) instruction. As memory known to be unused need not be examined, false positives and data ghosts are reduced.

In addition to a utilisation flag, call stack and channel flags are also recorded. Whenever a *call* instruction is executed, the workspace word used to store the return address is marked. This allows later reconstruction of the call stack of processes. Any word used as a channel in a channel input or output instruction is also marked to facilitate the building of process network graphs. If a process's workspace holds a channel word, or a pointer to a channel word, then it can be assumed it is a communication partner on that channel. When more than two such relationships exist then pointers can be assumed to represent the most recent partners on the channel, for example a channel declared in a parent process being used by two processes running in parallel beneath it.

```
MOBILE []BYTE data:
INT errno:
SEQ
  ... load bytecode into data array ...
  vm[request]  ! decode.bytecode; data
  vm[response] ? CASE
    CT.BYTECODE! bytecode:
    bytecode; bytecode
      -- bytecode decoded
      SEQ
        bytecode[request]  ! create.vm
        bytecode[response] ? CASE
          CT.VM.CTL! vm.ctl:
          vm; vm.ctl
            -- VM instance created
          error; errno
            ... handle VM creation error, e.g. out of memory ...
    error; errno
      ... handle decoding error, e.g. invalid bytecode ...
```

Figure 71: Decoding bytecode and creating a virtual machine instance for it.

5.7 A Tracing Debugger

The virtual machine extensions and interfaces so far detailed can be used to implement various development and debugging tools. This section details a tracing debugger.

5.7.1 Instancing Bytecode

First the bytecode we intend to run must be decoded, this is done by making a `decode.bytecode` request on the introspection channel. Having successfully decoded bytecode we can request a virtual machine instance to execute it. The code example in figure 71 demostrates decoding bytecode and creating a virtual machine instance. A virtual machine introspection bundle is assumed to be defined as `vm`.

5.7.2 Executing Bytecode

Before we can execute bytecode in our newly created virtual machine instance we must supply the parameters for the top-level-process. In a standard occam-pi program these

```
CT.CHANNEL? kyb.scr, scr.svr, err.svr: -- server ends
CT.CHANNEL! kyb.cli, scr.cli, err.cli: -- client ends
SEQ
  ... allocate channels ...
  -- fork off handling processes
  FORK keyboard.handler (kyb.cli, ...)
  FORK screen.handler (scr.svr, ...)
  FORK error.handler (err.svr, ...)
  -- set top level parameters
  vm.ctl[request]  ! set.param.chan; 0; kyb.scr
  vm.ctl[response] ? CASE ok
  vm.ctl[request]  ! set.param.chan; 1; scr.cli
  vm.ctl[response] ? CASE ok
  vm.ctl[request]  ! set.param.chan; 2; err.cli
  vm.ctl[response] ? CASE ok
```

Figure 72: Setting up a virtual machine top-level-process.

provide access to the keyboard input, and standard output and error channels. In this example tracing code we will assume the program being processed has a standard top-level-process; however, to handle alternate parameter combinations the `get.tlp` (section 5.5.4.5) request would be used to query the bytecode. In figure 72 we supply the top level parameters as channel bundles, preparing the bytecode for execution.

Having provided the top level parameters we can begin executing instructions from the bytecode. We want to trace the entire execution of the program thus we use the `step` request on the VM control channel bundle. If we did not require information about the execution of every instruction then we could use the `run` request. The code snippet in figure 73 demonstrates the `step` request, which returns two responses, one on instruction decode and one on dispatch.

5.7.3 Tracking Processes

We want to create a higher level view of the program execution than the raw instruction stream as it executes. Ideally we want to know the present line of execution of all the processes. To do this we need to track the executing processes and attribute instruction

```
IPTR iptr:
ADDR wptr:
INT op, arg:
SEQ
  vm.ctl[request]  ! step
  vm.ctl[response] ? CASE decoded; iptr; op; arg
  -- decoded instruction "op", with argument "arg"
  --   new instruction pointer: iptr
  vm.ctl[response] ? CASE dispatched; iptr; wptr
  -- dispatched instruction
  --   new instruction pointer: iptr
  --   new workspace pointer: wptr
```

Figure 73: Dispatching instruction on virtual machine.

execution to them. We need to monitor *start process* and *end process* instructions, additionally we must track any instruction that alters the workspace pointer (stack) such as *call* and *adjust workspace pointer*.

The code snippet in figure 74 converts the decoding and dispatching of instructions into a stream of `executing`, `start.process`, `end.process` and `rename.process` tagged protocol messages. Each message carries the process workspace, which acts as the process identifier. The `rename.process` messages indicate a change of process identifier, or a shift in workspace pointer.

5.7.4 Visualisation

The bytecode decoder interface detailed in section 5.5.4 is used to look up the present source position of each process. This allows us to generate output on the current source file and line being executed. Visualising this information as a graph of active processes we can see the executing program, although the specifics of this visualisation is an area for future work. Call graphs can be generated by monitoring *call* and *return* instructions, and by recording the process which executes a *start process* instruction as the parent of the process started, we can see the program structure.

Adding to the graph of active processes, we also build a graph of relationships between processes. Whenever a channel communication instruction is executed we record

```
ADDR id:
IPTR iptr:
WHILE TRUE
  ADDR new.id:
  SEQ
    -- report present process position
    out ! executing; id; iptr
    -- start process and end process
    vm.ctl[request]  ! step
    vm.ctl[response] ? CASE decoded; iptr; op; arg
    IF
      op = INS.OPR
        VM.STATE state:
        SEQ
          vm.ctl[request]  ! get.state
          vm.ctl[response] ? CASE state; state
          CASE arg
            INS.STARTP
              -- process workspace is on the operand stack
              out ! start.process; state[stack][0]
            INS.ENDP
              out ! end.process; id
            ELSE
              SKIP
       TRUE
         SKIP
     -- workspace pointer altering operations
     vm.ctl[response] ? CASE dispatched; iptr; new.id
     IF
       (op = INS.ADJW) OR (op = INS.CALL)
         out ! rename.process; id; new.id
       (op = INS.OPR) AND (arg = INS.RET)
         out ! rename.process; id; new.id
       TRUE
         SKIP
     -- update workspace pointer
     id := new.id
```

Figure 74: Implementing tracing of virtual machine instructions.

the channel pointer in a channel table. Processes which access the same channel become related; edges can be drawn between them in the graph of process nodes. On *adjust workspace pointer* instructions, and other memory deallocation instructions channel words which fall out of scope are deleted from the channel table. This information can be used to visualise the relationships between the active processes, either in real-time as communication occurs, or statically as a cumulative image recorded over time.

Figure 75 applies these techniques to generate an idealised visualisation of an example from the KRoC compiler distribution, `commstime.occ`. Curved boxes represent processes. Squared boxes represent calls in the call stack. Bold text is the present line being executed, as taken from the source file. Dots are channels, with arrowed lines representing communication requests. Hollow arrows are previously executed input or output, and filled arrows are active blocked requests.

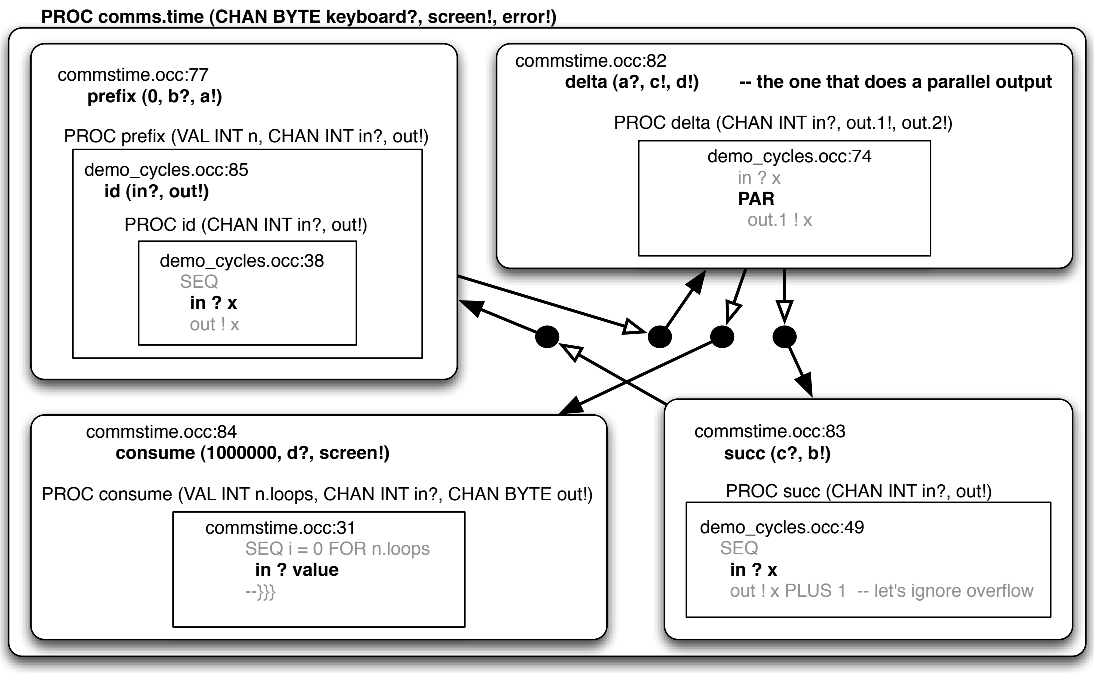

PROC comms.time (CHAN BYTE keyboard?, screen!, error!)
commstime.occ:77
prefix (0, b?, a!)
PROC prefix (VAL INT n, CHAN INT in?, out!)
demo_cycles.occ:85
id (in?, out!)
PROC id (CHAN INT in?, out!)
demo_cycles.occ:38
SEQ
in ? x
out ! x
commstime.occ:82
delta (a?, c!, d!)
-- the one that does a parallel output
PROC delta (CHAN INT in?, out.1!, out.2!)
demo_cycles.occ:74
in ? x
PAR
out.1 ! x
commstime.occ:84
consume (1000000, d?, screen!)
PROC consume (VAL INT n.loops, CHAN INT in?, CHAN BYTE out!)
commstime.occ:31
SEQ i = 0 FOR n.loops
in ? value
--}}}
commstime.occ:83
succ (c?, b!)
PROC succ (CHAN INT in?, out!)
demo_cycles.occ:49
SEQ
in ? x
out ! x PLUS 1 -- let's ignore overflow

Figure 75: Visualisation of commstime.occ.

5.8 Conclusions

A method for occam-pi programs to inspect and intercede on the execution of other occam-pi programs has been presented. This access is driven through a channel based interface; which permits reasoning about programs which use it, in-line with that applicable to a standard occam-pi program. Using this new interface it is possible to develop debugging tools for occam-pi using occam-pi, which allows us to use a full process-oriented programming environment in order to tackle the challenges faced developing such tools.

By constraining the probe effect caused by non-determinism related to the use of time in programs, cyclic debugging is permitted: the process of repeated execution of a program with slight modifications to its composition of inputs in an attempt to locate and fix an error. Additionally, memory typing information is exposed in order to help build complete debugging tools.

Virtual machine sub-instances presently run synchronously to their parents, a logical enhancement to our work is the multiplex execution of virtual machine instances. Further to this, by applying algorithms developed for the CCSP KRoC runtime, it should be possible to extend the TVM to execute multiple instances concurrently on multi-core shared-memory hardware. These concurrently executing instances will be able to communicate, wait and synchronise on each other safely while maximising use of any underlying hardware parallelism.

Given the method tool for inspecting the behaviour of running parallel programs developed here it would be ideal to make this power available in an easy to interpret graphical form. Exton identified four main areas of importance in the visualisation of program execution: abstraction, emphasis, representation and navigation [103]. Being able to provide a good abstraction level for observation of run-time activity, and the ability to adjust the level in tune with the user's expertise is critical to sensibly observing executing networks of processes. The design of a suitable visual model to represent the state of a running occam-pi program is an as yet unexplored area, and may potentially provide useful insight to inform related work in visual design of parallel programs.

Dynamic network topologies such as those possible with occam-pi add another dimension of complexity to the debugging process beyond that encountered with occam running on the Transputer. Dijkstra states that “our powers to visualise processes evolving in time are relatively poorly developed” [92], a statement that encapsulates some of the difficulty encountered when trying to reason about systems using dynamic process creation and mobile channels. Work to allow programmers of concurrent and process-oriented software to explore the execution of their program and interact with its run-time behaviour to diagnose problems and mistakes is an area still requiring much research.

Chapter 6

Conclusions and Further Work

This chapter summarises the broad conclusions of this thesis and a range of areas for further work.

6.1 Overview

This thesis has surveyed support for programming and managing concurrency in programming languages. A significant finding is that popular languages lack much support for concurrent programming, and that most support for concurrency programming primitives in common programming languages are data-oriented. This focus on a shared-memory model contrasts with a trend in hardware toward distributed and asymmetric models of computation. A focus on data-oriented methods also ignores complexities the data-oriented paradigm introduces through abstraction (see section 2.2.1).

Notationally, message-passing concurrency in general and process-oriented programming specifically has a high degree of mechanical sympathy with distributed memory computer systems. This is exemplified by the use of MPI in supercomputing applications. However message-passing usage tends to be limited to communication between computation nodes, and is not used between software components on the same node.

The technical work of this thesis demonstrates that process-oriented programming can be used for software structure *and* be efficient on modern multicore computers. This uses performance-centric approaches such as work-stealing, wait-free algorithms, cache affine batching and meticulous optimisation to reduce the overhead of all critical paths. The mechanical sympathy of process-oriented design is proven as unmodified software codes scale with the addition of processors. This is because process-oriented software provides parallel execution potential which can be efficiently extracted and utilised when synchronisation and context switch overheads are sufficiently amortized.

Additional technical work shows approaches for compiling and debugging process-oriented software. The work on compilation techniques demonstrates that the process-oriented approach is well supported by a continuation passing style of compilation (4.5) and modern architecture independent assembly languages such as LLVM. Recent work on light-weight context switching support for LLVM [97] and the development of continuation passing C [151] has also shown continuation passing style to be useful abstraction for concurrency in software. Presented work on introspection and debugging, while low-level in nature, suggests that a process-oriented programming style of concurrency is useful for reflection on concurrency itself in programming languages. Process-oriented reflection is a significant area for further investigation.

6.2 Specific Improvements

Definitions are given of process-oriented decompositions for common concurrent programming primitives in section 2.4.1. These definitions are informal, therefore it may be appropriate to supplement them with formal definitions and models. Work has already been done in this area by Welch and Barnes with models of barriers [255].

The algorithms contributed in this thesis rely on a total store order memory architecture and certain types of atomic operation such as *compare and swap*. Thus a useful area for further work is to explore implementing the same algorithms on different architectures and with *load-linked/store-conditional* atomics. An obvious candidate architecture is ARM, which is used in the majority of mobile phones and tablet computers. Most changes applicable to ARM are also applicable to IBM POWER which is used in large super-computing applications. Rather than simply emulating *compare and swap* and adding additional memory barrier instructions, optimal performance is likely to be achieved by restructuring the order of operations for greater sympathy with the hardware.

As previous noted, as the number of processor cores increases, most multiprocessor architectures develop *non-uniform memory architecture* (NUMA) characteristics. To allow increased scalability to this class of machine, there is a need to add NUMA-awareness to our scheduler and memory management subsystems and reassess the use of atomically mutable bitmaps of active schedulers, as these disregard locality. One promising solution is to dynamically constrain subsets of the process network and their associated batches to groups of logical processors. Random work stealing may not be the most appropriate for larger NUMA systems. The work stealing algorithms should be informed by the cache hierarchy to allow, for example, stealing of work which is in caches shared between adjacent cores.

For stack (rather than heap) languages it may be desirable to be able to dynamically grow process stacks. With occam-pi this is not necessary, but if it were then the minimum process size would be multiple pages rather than 64 bytes. On a 64-bit architecture this could be supported with a minimum process stack size of just a single page by using additional address space. However if process migration is supported (see 6.3) then the same mechanisms could be used for relocating and coalescing the memory of processes.

As a whole, memory management within the runtime is not addressed in this thesis; however, the cost of memory management can significantly affect the performance of runtime operations [148]. The scope of memory management required by occam-pi is sufficiently limited that it can be managed with reference counting alone. In modern languages a garbage collection scheme may be required, although the read-only nature of shared data in process-oriented software may reduce the complexity. Little work has been done on memory management for process-oriented software; the most comparable language is Manticore [108]. This is a clear area for further work.

6.3 Heterogeneous Architectures

Modern computer architectures are moving beyond simple NUMA, to systems with processors that vary distinctly in capability and speed. GPGPU is the most obvious example (2.5.14), where the graphics processor has its own distinct memory and instruction set. Graphics processor cores are moving toward general computation models and are increasingly integrated on the same physical package as other computational resources [224]. In order to support these code for co-processors must be compiled separately to produce machine code for each different type of processor involved. With present programming techniques this occurs once at compile time – components to run on the GPU are separated out where they are compiled into a different format. These are then loaded on to the GPU as needed.

There are a number of ways process-oriented programming can support heterogeneous architectures. One of the simplest is encapsulation, where a process abstracts and hides a piece of hardware. The hardware interface is made to appear as channel communication with a process. This technique is useful for special purpose hardware such as video decoders [219]. However it does not map well to general purpose computation.

Alternatively processes could be placed on different processors at creation time depending on the type of computation resources they require. Communication channels provide interaction between components on distinct processors. This partitioning is in principle the same as the current GPGPU programming techniques, although GPU computation is generally placed into a work queue from where it is dispatched. GPU computation in particular is grouped into jobs which are scheduled by the hardware (or hardware driver). Thus an alternative would be *process coalescing*. In occam-pi a loop can be declared parallel, in which case the processing of each loop element is undertaken by a separate concurrent process:

```
PAR i = 0 FOR SIZE data
  data[i] := compute.new.value(data[i])
```

Each process is independent but conducting the same computation. Assuming no communication, this computation is an ideal candidate for vectorisation. Work by Damian Dimmich has looked at manual vectorisation of these forms [95]. For automatic vectorisation, groups of similar processes could be coalesced and executed as a single instruction sequence on a SIMD processor such as a GPU. This approach is essentially stream programming [62, 156]. By applying aspects of stream programming research, process networks and communication within them could be converted to SIMD vector instruction streams.

When general purpose computing facilities are available, *process migration* rather than explicit placement is ideal. A process provides a unit of data and computation that should be relocatable within a larger system. If mutable shared data is disallowed then the closure of a process can be relocated as long as its communication links are maintained. Within a system of shared memory (even heavily distributed) then versions of the algorithms presented in this thesis could be used to support communication. In larger systems, a network based approach may be necessary. Previous work by Mario Schweigler has shown how occam-pi channels can be automatically reconnected across distributed networks [230]. This also works for dynamic process networks.

Previous work with occam-pi and mobile processes has focused on explicit control of when a process is running or stopped [257]. This is a useful low level behaviour, however to provide a useful high-level abstraction the movement of processes within a system should be transparent and automatic. The programmer could add hints to their code about what processing facilities a process would benefit from and thus influence its placement. Additionally, the compiler could perform a similar computation of hints. The placement of processes on appropriate hardware is then a dynamic best-effort constraint solving exercise.

Typically the code used to process data is smaller than the data being processed. Thus for a distributed process-oriented system the process should be moved close to the data (or communication parties it engages with frequently) rather than moving the data across the system [176]. To facilitate this scenarios *process migration* must be computationally cheap. This is not true for large operating system processes, but is a possibility for process-oriented fine-grain concurrency [187].

In summary, runtime support (and compilation) for process-oriented programs should, going forward, focus on *process migration* and *process coalescing*. This will allow process-oriented software (potentially unmodified) to be applicable to evolving hardware in both embedded systems (mobile phones and tablets), desktop computers and super-computing applications.

6.4 Runtime Embedding

The significance of the Internet and the World Wide Web cannot be overlooked. Necessitated by the lack of concurrent programming models, development of web applications, both client-side and server-side uses an event-driven model. On the server-side, incoming requests are distinct events which start a new computation and manipulate shared data. Persistent state must be emulated by retrieving it at the beginning of a request or encapsulating it in the request itself. On the client side, the application makes requests of servers and sets up event handlers for them (and other events such as user button clicks).

While an asynchronous communication model is appropriate for requests communicated over the internet, it may not be appropriate at the application level. Event-driven programming obscures control flow [151]. Thus work should be done on how process-oriented style programming can be supported in this space. For example using web browser plugins such as Chrome Native Client to provide alternative programming enviroments or extensions to JavaScript which provide runtime support for concurrency. It is possible to see that with appropriate extensions and runtime support JavaScript could be used as a compilation target for a process-oriented language, much as is done with Clojure (2.5.9). Research is required in this area, not least to investigate the efficiency of such an approach on power constrained mobile devices.

For server programming, integration with existing server architectures to support persistent light-weight processes is foreseeable. Akka (2.5.26.1) with its light-weight Actor model concurrency has found some popularity in this space. Work needs to be carried out to investigate whether WebSockets can be used for communication between process-oriented server and client components [132]. An ideal scenario is a synchronous efficient process-oriented application running on both client and server, communicating over the asynchronous Internet in a consistent manner. Persistent components of the server system could be distributed amongst an array of machines in a cloud environment with process migration occurring to rebalance load dynamically at run-time.

6.5 New Languages

The technical work of this thesis predates or overlaps the release of new programming languages with process-oriented concurrency models, notably Google Go (2.5.13) and Rust (2.5.25). Broadly speaking the techniques developed in this thesis should be applicable to these languages. Practical work is required to prove this.

For Google Go, mutable data and race hazards in the select (choice) statement present two implementation concerns. A "safe" subset of the language could be defined, although the use of features in the standard library would need to be examined. As the select statement offers no guarantees of ordering, the alternation algorithm need only be modified to maintain correctness of implementation when multiple processes alternate on the same channel, rather than guaranteeing a specific behaviour (with respect to commitment). Implicit locking of shared channels is also required. Alternatively, a heavy-weight generalised ALT implementation such as that of Lowe could be used [165].

Rust has a memory model based on isolation, where mutable state is encapsulated in a manner similar to occam-pi's mobile data model. Additionally the choice primitive, select, is only applicable to input. This asymmetry matches occam-pi's alternation which only permits choice on input channel ends. The absence of shared memory allows Rust to use a per-task garbage collection model for memory management. Thus from a runtime perspective Rust is a good candidate for further exploration of the algorithms presented in this thesis. From a language perspective, Rust's heavy use of symbols on top of its C derived syntax may act to hinder its adoption.

Beyond these examples there is still scope for the development of new languages based on a strongly process-oriented model of concurrency. One aspect overlooked by Google Go in particular, and Rust to a lesser extent is compile time and run-time analysis to avoid common concurrency errors. Compile time analysis and the embedding of known safe patterns in the language design can greatly reduce programmer errors. Any new process-oriented language should by default generate models appropriate for formal reasoning about the interactions of components [228]. Additionally, run-time analysis may be required for dynamic process networks. The programmer should be guided towards, and where necessary constrained to, safe and mechanically sympathetic models of computation which are accessible to both informal and formal reasoning.

6.6 Closing Remarks

Driven by physical limits and resource constraints, computer architectures are increasingly parallel and heterogeneous. Process-oriented programming is well-equipped to both manage *and* utilise the concurrency present in these new and emerging computer systems. The technical contributions of this thesis provide the engineering required to realise the potential of fine-grain process-oriented programming on modern multi-core hardware. Software written in a process-oriented style today should, through further work on runtime systems, be equally applicable to the computer systems of tomorrow.

Appendix A

Runtime Interface

This appendix provides an overview of the standardised interface to the runtime kernel developed to support occam-pi compilation.

A.1 Calling Convention

Kernel calls use a modified C calling function designed to keep as much information in registers such as to avoid stack access. Only Intel x86 (32-bit) is presently supported and hence that is the only calling convention documented here. Importantly on entering a kernel call two pieces of information are always made available: the logical processor data structure pointer (`sched`), and the workspace pointer of the present process (`Wptr`). How these are managed outside the kernel is not important, but they must be passed in with every kernel call.

A.1.1 tranx86

For *tranx86* compiled code the kernel functions are defined (in C) as:

```
void __attribute__ ((regparm(3)))
```

```
kernel_symbol
(word param0, sched_t *sched, word *Wptr)
```

This means the first parameter passed to the kernel call is in `%EAX`, the scheduler pointer in `%EDX` and the workspace pointer in `%ECX`. Remaining inputs are stored on the stack. The kernel uses the return address stored on the stack.

On exiting a runtime kernel call, any output is returned in `%EAX`. If a context switch has not occurred then the stack will be unwound in C. If the return instruction pointer needs to be changed, then this will be done before the stack is unwound so that the `RET` instruction jumps back to the new instruction pointer. If a context switch has occurred then an assembly instruction sequence is used to reset the stack pointer, losing all stack frames allocated since kernel entry, and then jump to the new instruction pointer. The workspace pointer must also be restored to the *tranx86* register used for workspace pointers `%EBP`.

```
movl Wptr        , %ebp
movl sched->stack, %esp
jmp *-4(%ebp)
```

A.1.2 LLVM

LLVM kernel call functions use standard C calling convention. Thus LLVM kernel calls are defined as:

```
word *kernel_Y_symbol(sched_t *sched, word *Wptr, ...)
word kernel_X_symbol(sched_t *sched, word *Wptr, ...)
```

Context switching functions return a new `Wptr` as a `word *` type. Other functions return a `word` value. This means that context switching kernel calls cannot return values, which disables some special case alternation calls, but otherwise is compatible with the *tranx86* runtime. On returning from a kernel call it is the responsibility of the generated

LLVM code to tail call the continuation pointed to by `Wptr[Iptr]` passing the `sched` and `Wptr` values to it.

A.2 Processes

A.2.1 endp

Symbol:	Y_endp
Mnemonics:	**ENDP**
Input:	`Wptr`

End current process, potentially completing a process barrier. This passes the `Wptr` of the parent process which contains a process barrier. An atomic decrement and test is taken on the barrier (at `Wptr[Count]`) and if the count reaches zero, `Wptr` is update (saved priority, affinity and instruction pointer restored) and placed on a run-queue.

A.2.2 mreleasep

Symbol:	Y_mreleasep
Mnemonics:	**MRELEASEP**
Input:	`adjust`

Frees a process whos workspace was allocated by `X_malloc`. The workspace pointer of the calling process is adjusted by `adjust` words, and then released as a mobile type. This supports recursive calls (which allocated new workspaces).

A.2.3 pause

Symbol:	Y_pause
Mnemonics:	**Only used during runtime start.**

Reschedule by placing the current process on run-queue and entering scheduler loop.

A.2.4 par_enroll

Symbol:	X_par_enroll
Mnemonics:	**PAR_ENROLL**
Input:	`Wptr`, `count`

Enrols processes on a process barrier. This performs an atomic increment on `Wptr[Count]` of `count`.

A.2.5 proc_alloc

Symbol:	X_proc_alloc
Mnemonics:	**PROC_ALLOC**
Input:	`flags`, `words`
Output:	`Wptr`

Allocate a process workspace of `words` size (in machine words). The `flags` are not used. The returned workspace is a mobile type.

A.2.6 proc_mt_copy

Symbol:	X_proc_mt_copy
Mnemonics:	**PROC_MT_COPY**
Input:	`Wptr`, `offset`, `mobile type`

Copy a mobile type to workspace allocated via `X_proc_alloc`. Install a mobile type parameter into a workspace, this is equal to `Wptr[offset]:=mt_clone(mobile type)`. This exists to facilitate the creation of workspaces in distributed memories.

A.2.7 proc_mt_move

Symbol:	X_proc_mt_move
Mnemonics:	**PROC_MT_MOVE**
Input:	`Wptr, offset, mobile type pointer`

Move a mobile type to workspace allocated via `X_proc_alloc`. Install a mobile type parameter into a workspace; this is equal to `Wptr[offset] := *pointer`. The callers reference to the mobile type is set to `null`. This exists to facilitate the creation of workspaces in distributed memories.

A.2.8 proc_param

Symbol:	X_proc_param
Mnemonics:	**PROC_PARAM**
Input:	`Wptr, offset, parameter`

Pass a param to workspace allocated via `X_proc_alloc`. Install a parameter into a workspace; this is equal to `Wptr[offset] := parameter`. This exists to facilitate the creation of workspaces in distributed memories.

A.2.9 proc_start

Symbol:	Y_proc_start
Mnemonics:	**PROC_START**
Input:	`offset, Wptr, Iptr`

Start a process using a workspace allocated via `X_proc_alloc`. The `Wptr` is adjusted by `offset` words and made into a valid process descriptor before being placed on a run-queue. The adjustment is required as workspaces grow down, but the allocated `Wptr` is for the bottom of the workspace.

A.2.10 runp

Symbol:	X_runp
Mnemonics:	**RUNP**
Input:	`Wptr`

Adds a process to run-queue. The process descriptor `Wptr` is added to an appropriate run-queue and then control returns to the calling process. This constrasts with `Y_startp` as a context switch will not occur. The relevant fields of `Wptr` such as `Iptr` and `Priofinity` must be made valid before this call.

A.2.11 startp

Symbol:	Y_startp
Mnemonics:	**STARTP**
Input:	`Wptr, Iptr`

Start a process. `Wptr` is the initial workspace of the process and `Iptr` the initial instruction pointer. The process inherits the priority and affinity of the current process. The runtime will either place the new process on a run-queue for later execution or perform a context switch to it immediately. In either case the calling process may be switched out. This is a change from the `STARTP` behaviour of the Transputer [139].

A.2.12 stopp

Symbol:	Y_stopp
Mnemonics:	**STOPP**

Stop current process. Make the process descriptor valid (for use with `X_runp`) and enter scheduler loop. Note the process descriptor is not placed on a scheduler loop.

A.3 Mobiles

A.3.1 malloc

Symbol:	X_malloc
Mnemonics:	**MALLOC**
Input:	`size`
Output:	`mobile type pointer`

Allocates memory of a given size. Produces a mobile type if size $>$ 0, otherwise returns `null`.

A.3.2 mrelease

Symbol:	X_mrelease
Mnemonics:	**MRELEASE**
Input:	`mobile type pointer`

Releases memory from `X_malloc`. This is the same as `X_mt_release`.

A.3.3 mt_alloc

Symbol:	X_mt_alloc
Mnemonics:	**MT_ALLOC**
Input:	`type`, `size`
Output:	`mobile type pointer`

Allocates a new mobile type and initialises it. See appendix B.

A.3.4 mt_bind

Symbol:	X_mt_bind
Mnemonics:	**MT_BIND**
Input:	`bind type, mobile type pointer, data pointer`
Output:	`mobile type pointer`

Bind a mobile type in some way to a bit of data. This is used by RMoX to bind mobiles to DMA capable memory or between virtual and physical spaces. Returns a replacement mobile type pointer as a result.

A.3.5 mt_clone

Symbol:	X_mt_clone
Mnemonics:	**MT_CLONE**
Input:	`mobile type pointer (of source)`
Output:	`mobile type pointer (of clone)`

Clones a mobile type.

A.3.6 mt_dclone

Symbol:	X_mt_dclone
Mnemonics:	**MT_DCLONE**
Input:	`type, size, pointer`
Output:	`mobile type pointer`

Clones some data into a new mobile type. This allocates a mobile type and copies in `size` bytes from `pointer`.

A.3.7 mt_enroll

Symbol:	X_mt_enroll
Mnemonics:	**MT_ENROLL**
Input:	`count, mobile type`

Resign on a mobile type. Increase the barrier enrollment count by `count`.

A.3.8 mt_lock

Symbol:	Y_mt_lock
Mnemonics:	**MT_LOCK**
Input:	`lock type, mobile type`

Lock a mobile type. The type of lock is specified by `lock type`, which can be `MT_CB_CLIENT` or `MT_CB_SERVER` which represent the two directions a channel can be locked in (these use separate semaphores internally). Calling process is blocked until the lock is gained.

A.3.9 mt_release

Symbol:	X_mt_release
Mnemonics:	**MT_RELEASE**
Input:	`mobile type pointer`

Frees a mobile type.

A.3.10 mt_resign

Symbol:	X_mt_resign
Mnemonics:	**MT_RESIGN**
Input:	`count, mobile type`

Resign from a mobile type. Reduce the barrier enrollment count by `count`.

A.3.11 mt_resize

Symbol:	X_mt_resize
Mnemonics:	**MT_RESIZE**
Input:	`resize type`, `mobile type pointer`, `argument`
Output:	`mobile type pointer`

Resize a mobile type. Only a `resize type` of `MT_RESIZE_DATA` is supported. This increases or decreases the size of the mobile type so it can hold at least `argument` bytes. Returns a replacement mobile type pointer as a result.

A.3.12 mt_sync

Symbol:	Y_mt_sync
Mnemonics:	**MT_SYNC**
Input:	`mobile type`

Synchronise on mobile type. This is used for barriers in which case the call will not complete until the barrier completes.

A.3.13 mt_unlock

Symbol:	X_mt_unlock
Mnemonics:	**MT_UNLOCK**
Input:	`lock type`, `mobile type`

Unlock a mobile type. The type of lock is specified by `lock type`, which can be `MT_CB_CLIENT` or `MT_CB_SERVER` which represent the two directions a channel can be locked in (these use separate semaphores internally). This assumes the lock was previously gained with `Y_mt_lock`.

A.4 Channels

A.4.1 xable

Symbol:	Y_xable
Mnemonics:	**XABLE**
Input:	`channel`

Synchronise with outputting process on `channel`. The caller is descheduled until an output process arrives on the channel. This should be followed by an `X_xend`.

A.4.2 xend

Symbol:	X_xend
Mnemonics:	**XEND**
Input:	`channel`

Reschedules outputting process blocked on `channel`. This should follow a `Y_xable`.

A.4.3 in

Symbol:	Y_in
Mnemonics:	**IN**
Input:	`count`, `channel`, `pointer`

Read `count` bytes from `channel` into `pointer`.

A.4.4 in32

Symbol:	Y_in32
Mnemonics:	**IN32**
Input:	`channel`, `pointer`

Read 4 bytes from `channel` into `pointer`.

A.4.5 in8

Symbol:	Y_in8
Mnemonics:	**IN8**
Input:	`channel`, `pointer`

Read 1 byte from `channel` into `pointer`.

A.4.6 xin

Symbol:	Y_xin
Mnemonics:	**XIN**
Input:	`count`, `channel`, `pointer`

Read `count` bytes from `channel` into `pointer`. Does not reschedule the process blocked on the channel after it completes. This should follow a `Y_xable` and be followed by a `X_xend`.

A.4.7 mt_in

Symbol:	Y_mt_in
Mnemonics:	**MT_IN**
Input:	`channel`, `pointer`

Read a mobile type from `channel` and store it in `pointer`. This updates the reference count if the mobile type is a copy type such as shared channel. This enrolls the receiver if the mobile type was a barrier.

A.4.8 mt_out

Symbol:	Y_mt_out
Mnemonics:	**MT_OUT**
Input:	`channel`, `mobile type pointer`

Writes a mobile type to `channel`. This updates the reference count if the mobile type is a copy type such as shared channel. If the type is a moving type, for example mobile data, then the pointer is made `null` on leaving this call.

A.4.9 mt_xchg

Symbol:	Y_mt_xchg
Mnemonics:	**MT_XCHG**
Input:	`channel, mobile type pointer`

Swaps a mobile type via `channel`. After the call the `mobile type pointer` will point at the other communicating party's mobile type.

A.4.10 mt_xin

Symbol:	Y_mt_xin
Mnemonics:	**MT_XIN**
Input:	`channel, pointer`

Read a mobile type pointer from the `channel` and store it in `pointer`. This updates the reference count if the mobile type is a copy type such as shared channel. This enrolls the receiver if the mobile type was a barrier. Does not reschedule the process blocked on the channel after it completes. This should follow a `Y_xable` and be followed by a `X_xend`.

A.4.11 mt_xout

Symbol:	Y_mt_xout
Mnemonics:	**MT_XOUT**
Input:	`channel, mobile type pointer`

Writes a mobile type to `channel`. This updates the reference count if the mobile type is a copy type such as shared channel. If the type is a moving type, for example mobile data, then the pointer is made `null` on leaving this call. Does not reschedule the process blocked on the channel after it completes. This should follow a `Y_xable` and be followed by a `X_xend`.

A.4.12 mt_xxchg

Symbol:	Y_mt_xxchg
Mnemonics:	**MT_XXCHG**
Input:	`channel, mobile type pointer`

Swaps a mobile type via the `channel`. After the call the `pointer` will point at the other communicating party's mobile type. Does not reschedule the process blocked on the channel after it completes. This should follow a `Y_xable` and be followed by a `X_xend`.

A.4.13 out

Symbol:	Y_out
Mnemonics:	**OUT**
Input:	`count, channel, pointer`

Write `count` bytes from `pointer` into `channel`.

A.4.14 out32

Symbol:	Y_out32
Mnemonics:	**OUT32**
Input:	`channel, pointer`

Write 4 byte from `pointer` into `channel`.

A.4.15 out8

Symbol:	Y_out8
Mnemonics:	**OUT8**
Input:	`channel`, `pointer`

Write 1 byte from `pointer` into `channel`.

A.4.16 outbyte

Symbol:	Y_outbyte
Mnemonics:	**OUTBYTE**
Input:	`value`, `channel`

Write the byte `value` to `channel`. The byte is stored in `Wptr[Temp]`.

A.4.17 outword

Symbol:	Y_outword
Mnemonics:	**OUTWORD**
Input:	`value`, `channel`

Write the word `value` to `channel`. The word is stored in `Wptr[Temp]`.

A.5 Alternation

A.5.1 alt

Symbol:	X_alt
Mnemonics:	**ALT**

Begin alternation. This sets up the process descriptor for alternation. Only other alternation calls should follow this until `Y_altend` is called.

A.5.2 altend

Symbol:	Y_altend
Mnemonics:	**ALTEND**

Finishes alternation. Unlike the T9000 variant this may reschedule the calling process to avoid races.

A.5.3 talt

Symbol:	X_talt
Mnemonics:	**TALT**

Begin timer alternation. This sets up the process descriptor for alternation. It also initialises timer related fields. Only other alternation calls should follow this until `Y_taltend` is called.

A.5.4 altwt

Symbol:	Y_altwt
Mnemonics:	**ALTWT**

Wait on alternation. This may return immediately if a guard has already become ready. Only other alternation calls should follow this until `Y_altend` is called.

A.5.5 taltwt

Symbol:	Y_taltwt
Mnemonics:	**TALTWT**

Wait on timer alternation, checking timer fields and setting up timer node if required. This may return immediately if a guard has already become ready. On return the `Time_f` will be set the time the process was rescheduled. Only other alternation calls should follow this until `Y_taltend` is called.

A.5.6 disc

Symbol:	X_disc
Mnemonics:	**DISC**
Input:	`ready Iptr`, `guard`, `channel`
Output:	`ready`

Disable channel. If `guard` is not zero disable `channel`. If the `channel` is ready *and* `Wptr[Temp]` is `null` then store the `ready Iptr` in `Wptr[Temp]` and return true. Otherwise return false.

A.5.7 diss

Symbol:	X_diss
Mnemonics:	**DISS**
Input:	`ready Iptr`, `guard`
Output:	`ready`

Disable SKIP guard. If `guard` is not zero *and* `Wptr[Temp]` is `null` then store the `ready Iptr` in `Wptr[Temp]` and return true. Otherwise return false.

A.5.8 dist

Symbol:	X_dist
Mnemonics:	**DIST**
Input:	`ready Iptr`, `guard`, `timeout`
Output:	`ready`

Disable timer. If `guard` is not zero *and* `timeout` is after `Wptr[Time_f]` *and* `Wptr[Temp]` is `null` then store the `ready Iptr` in `Wptr[Temp]` and return true. Otherwise return false.

A.5.9 enbc

Symbol:	X_enbc
Mnemonics:	**ENBC**
Input:	`guard`, `channel`
Output:	`ready`

Enable channel. If `guard` is not zero, then `channel` is enabled for alternation. Returns whether `channel` is ready (0 or 1).

A.5.10 enbc2

Symbol:	Y_enbc2
Mnemonics:	**ENBC2**
Input:	`ready Iptr`, `channel`

Enable `channel` with `ready` address. If the `channel` is ready then the call returns to the instruction pointer `ready Iptr`.

A.5.11 enbc3

Symbol:	Y_enbc3
Mnemonics:	**ENBC3**
Input:	`ready Iptr`, `guard`, `channel`
Output:	`ready`

Enable `channel` with `ready` address. If `guard` is not zero, then channel is enabled for alterntion. If the `channel` is ready then the call returns to the instruction pointer `ready Iptr`. Return value depends on if `channel` is ready (0 or 1).

A.5.12 enbs

Symbol:	X_enbs
Mnemonics:	**ENBS**
Input:	`guard`
Output:	`ready`

Enable skip guard. If `guard` is not zero then the alternation state is automatically made ready. The return value of `ready` is the same as `guard`.

A.5.13 enbs2

Symbol:	Y_enbs2
Mnemonics:	**ENBS2**
Input:	`ready Iptr`

Enable skip guard with `ready` address. The alternation state is automatically made ready. The call returns to the instruction pointer `ready Iptr`.

A.5.14 enbs3

Symbol:	Y_enbs3
Mnemonics:	**ENBS3**
Input:	`ready Iptr`, `guard`
Output:	`ready`

Enable skip guard with `ready` address and `guard`. If `guard` is not zero then the alternation state is automatically made ready. The call returns to the instruction pointer `ready Iptr`. The return value of `ready` is the same as `guard`.

A.5.15 enbt

Symbol:	X_enbt
Mnemonics:	**ENBT**
Input:	`guard`, `timeout`
Output:	`ready`

Enable timer. If `guard` is not zero, enable alternation timeout at `timeout`. If `timeout` has already passed then automatically make state ready. The timeout value is stored in the `Time_f` field; if this was already set then the closer value in time is used. Return value indicates if the timer is ready or not (0 or 1).

A.5.16 enbt2

Symbol:	Y_enbt2
Mnemonics:	**ENBT2**
Input:	`ready Iptr`, `timeout`

Enable timer with `ready` address. If `timeout` has already passed then automatically make state ready. The timeout value is stored in the `Time_f` field; if this was already set then the closer value in time is used. If ready the call returns to the instruction pointer `ready Iptr`.

A.5.17 enbt3

Symbol:	Y_enbt3
Mnemonics:	**ENBT3**
Input:	`ready Iptr`, `guard`, `timeout`
Output:	`ready`

Enable timer with `ready` address and `guard`. If `guard` is not zero, enable alternation timeout at `timeout`. If `timeout` has already passed then automatically make state ready. The timeout value is stored in the `Time_f` field; if this was already set then the closer value in time is used. Return value indicates if the timer is ready or not (0 or 1). If ready the call returns to the instruction pointer `ready Iptr`.

A.5.18 ndisc

Symbol:	X_ndisc
Mnemonics:	**NDISC**
Input:	`ready Iptr`, `guard`, `channel`
Output:	`ready`

Disable channel (new style). If `guard` is not zero disable `channel`. If the `channel` is ready then store the `ready Iptr` in `Wptr[Temp]` (overwriting any other value store there) and return true. Otherwise return false.

A.5.19 ndiss

Symbol:	X_ndiss
Mnemonics:	**NDISS**
Input:	`ready Iptr`, `guard`
Output:	`ready`

Disable SKIP guard (new style). If `guard` is not zero then store the `ready Iptr` in `Wptr[Temp]` (overwriting any other value store there) and return true. Otherwise return false.

A.5.20 ndist

Symbol:	X_ndist
Mnemonics:	**NDIST**
Input:	`ready Iptr`, `guard`, `timeout`
Output:	`ready`

Disable timer (new style). If `guard` is not zero *and* `timeout` is after `Wptr[Time_f]` then store the `ready Iptr` in `Wptr[Temp]` (overwriting any other value store there) and return true. Otherwise return false.

A.6 Auxiliary

A.6.1 getaff

Symbol:	X_getaff
Mnemonics:	**GETAFF**
Output:	`bitmap`

Get processor affinity. Returns the present affinity bitmap from the `priofinity` field of the logical processor.

A.6.2 getpas

Symbol:	X_getpas
Mnemonics:	**GETPAS**
Output:	`priofinity`

Get current raw priofinity. The unmodified value of the `priofinity` field of the logical processor is returned. This is can be stored in a process barrier or another process workspace to copy both affinity and priority settings.

A.6.3 getpri

Symbol:	X_getpri
Mnemonics:	**GETPRI**
Output:	`priority`

Get process priority. Return the present priority from the `priofinity` field of the logical processor.

A.6.4 proc_end

Symbol:	Y_proc_end
Mnemonics:	**PROC_END**
Input:	`Wptr`

Called by a process started by `Y_proc_start` to terminate. The `Wptr` is released (as a mobile type) and the scheduler loop entered. This is similar to `Y_mreleasep`; however, the valid Wptr is already known so an offset is not required.

A.6.5 sem_claim

Symbol:	Y_sem_claim
Mnemonics:	**SEM_CLAIM**
Input:	`pointer`

Claim semaphore at `pointer`. Calling process is blocked until the semaphore is claimed.

A.6.6 sem_init

Symbol:	X_sem_init
Mnemonics:	**SEM_INIT**
Input:	`pointer`

Initialise semaphore at `pointer`. The semaphore `fptr` and `bptr` words are set to represent an empty queue with the lock released.

A.6.7 sem_release

Symbol:	X_sem_release
Mnemonics:	**SEM_RELEASE**
Input:	`pointer`

Release semaphore at `pointer`. This assumes the semaphore was previously claimed with `Y_sem_claim`.

A.6.8 setaff

Symbol:	Y_setaff
Mnemonics:	**SETAFF**
Input:	`bitmap`

Sets the present affinity bitmap in the `priofinity` field of the logical processor. If necessary the present batch will be split to maintain the same priofinity invariant of batches. The process may be mailed to another logical processor if the bitmap no longer supports execution on the present logical processor.

A.6.9 setpri

Symbol:	Y_setpri
Mnemonics:	**SETPRI**
Input:	`priority`

Set process priority. If the priority of the current process is changed then it will be placed on an appropriate run-queue and scheduler loop entered.

A.6.10 wait_int

Symbol:	Y_wait_int
Mnemonics:	**WAIT_INT**
Input:	`interrupt, mask`

Entry point for an RMoX (occam-pi operating system) process to wait for an interrupt [46]. Special kernel call allowing processes to wait on processor interrupts. The `mask` is not used.

A.7 Other

A.7.1 BasicRangeError

Symbol:	Y_BasicRangeError

Entry point for range check error instruction with no information. Display appropriate error message and stop runtime.

A.7.2 dtrace

Symbol:	X_dtrace
Input:	`trapval_A`, `trapval_B`

Handles debugging traces generated by tranx86.

A.7.3 floaterr

Symbol:	Y_floaterr
Input:	`info2`, `info1`, `Wptr`, `filename pointer`, `FPU status`

Entry point for floating point error. Display appropriate error message and stop runtime.

A.7.4 fmul

Symbol:	X_fmul
Mnemonics:	**FMUL**
Input:	`Areg`, `Breg`
Output:	`Creg`

FMUL implementation. This is done in C as it is easier than assembly expansion. See `FMUL` instruction for T9000 [139].

A.7.5 norm

Symbol:	X_norm
Mnemonics:	**NORM**
Input:	`Areg`, `Breg`
Output:	`Areg`, `Breg`, `Creg`

Normalises 64-bit double-length integer. This is done in C as it is easier than assembly expansion. See `NORM` instruction FMUL [139].

A.7.6 overflow

Symbol:	Y_overflow
Input:	`info2`, `info1`, `Wptr`, `filename pointer`

Entry point for arithmetic overflow. Display appropriate error message and stop runtime.

A.7.7 RangeCheckError

Symbol:	Y_RangeCheckError
Input:	`info2`, `info1`, `Wptr`, `filename pointer`

Entry point for range check error instruction with no information. This does not take a return address as no return is expected. Display appropriate error message and stop runtime.

A.7.8 rtthreadinit

Symbol:	Y_rtthreadinit
Input:	`Wptr`

Call by logical processor thread at start up. Allocates and initialises logical processor data structures, installs a global pointer to itself and marks the logical processor enabled. If this is the first thread it then starts other runtime threads, up to count specified by `CCSP_RUNTIME_THREADS` environment variable or number of detected processors. It then begins executing the process specified by `Wptr` if it is not `null`, else enters scheduler loop to perform work stealing.

A.7.9 BNSeterr

Symbol:	Y_BNSeterr
Mnemonics:	**SETERR**

Entry point for SETERR instruction with no information, does not provide a return address. Display appropriate error message and stop runtime.

A.7.10 shutdown

Symbol:	Y_shutdown
Mnemonics:	**SHUTDOWN**

Sets shutdown flag in runtime and enters scheduler loop. All schedulers will eventually detect the flag and terminate.

A.7.11 trap

Symbol:	X_trap
Input:	`trapval_A`, `trapval_B`, `trapval_C`
Output:	`trapval_A`, `trapval_B`, `trapval_C`

Trap outputing present state of registers, workspace and stack. Supports continued execution.

A.7.12 unsupported

Symbol:	Y_unsupported

Unsupported kernel call. Displays an error message and shuts down runtime.

A.7.13 zero_div

Symbol:	Y_zero_div
Input:	`info2, info1, Wptr, filename pointer`

Entry point for integer division-by-zero error. Display appropriate error message and stop runtime.

Appendix B

Mobile Types

This appendix provides an overview of the mobile types object model developed to support compilation of occam-pi. Prior to this object model all mobile types were allocated in untyped memory; all type information was embedded in the compiler and was not accessible at runtime.

There are two different types of data in the occam-pi language, static data and mobile data. When sending static data over a channel it is copied from one process workspace to another; this contrasts with mobile data which is transferred by reference. The occam-pi compiler strictly manages the references to mobile data such that there is only one at a time; this prevents all race hazards at compile time. The runtime provides facilities for allocating, releasing, cloning and communicating mobile data. Additionally the runtime also provides the same facilities for channels, barriers and complex hierarchical structures containing other mobiles.

There are three types of mobile:

- *arrays* of basic data bytes and of other mobiles,
- *channel bundles*,
- and *barriers*.

Mobile channel bundles are arrays of channel words. They may be declared as shared, in which case their references are copied when communicated instead of moved. The references to shared mobile channel bundles are counted, release operations decrementing the count until it reaches zero. Additionally, shared channels contain a pair of semaphores which can be used to lock and unlock each direction of communication.

Barriers are reference counted in addition to their enrollment count. This allows a process to resign from a barrier while maintaining a reference to it for later use, e.g. re-enrolling. Whenever a barrier is communicated, the receiving process is automatically and atomically enrolled as part of the communication. This facilitates reasoning, as the sender may send a barrier then synchronise on it, safe in the knowledge that the synchronisation cannot complete until the receiver either synchronises or resigns.

B.1 Descriptors

Mobile types are described by bit-packed descriptors consisting of a machine word. The basic structure of a descriptor is shown in figure 76.

- *simple flag*: This flag bit is presently always set. It indicates that the type is solely contained in the descriptor word. In future clearing this bit will indicate the type descriptor word is a pointer to a descriptor structure in memory. This will allow programs to define complex mobile types containing various nest mobiles at runtime.

- *type number*: Four bits represent the mobile type number. See section B.2 for a complete list of types.

- *type flags*: These bits, zero or more, contain flags specific to the type number. Typically there will always be three type specific flag bits, rounding the size of a type descriptor up to at least a byte.

```
N            5             1             0 LSB
| type flags | type number | simple flag |
```

Figure 76: Mobile type descriptor layout.

The descriptor word is stored in the first memory word below the mobile type memory, i.e. `((word *)mobile)[-1]`. This allows all mobile types to be self contained allocation. The run-time may store additional data within the mobile type; however, any such data is implementation specific. Only the type descriptor should be ever be relied on.

B.2 Type Numbers

B.2.1 0: Numeric Data

The mobile type is numeric data in machine byte-order, the type flags $(0-2)$ represent a numeric identifier for the subtype:

0	BYTE
1	INT16
2	INT32
3	INT64
4	REAL32
5	REAL64

These subtypes are arrange so that the number of bytes required to represent them can be calculated from the type flag bits as follows: $2^{(\text{flag}_2+(\text{flag}_1\times 2)+\text{flag}_0)}$.

This can be simply expressed in C syntax as:

```
2 << ((flags >> 2) + (flags & 0x3))
```

Any remaining bits in the mobile type descriptor word are not used.

```
MSB                                               LSB
 8                5                  1                0
 | dimensions: 1   | mobile array (1) | simple flag (1) |

 16               13                 9                8
 | numeric type: 2 | numeric data (0) | simple flag (1) |
```

Figure 77: Example mobile array type descriptor.

B.2.2 1: Mobile Array

The type flags $(0-2)$ of the mobile array represent the number of dimensions minus one. Hence if the type flags are zero then the array has one dimension. In memory the array is represented as:

```
struct mt_array_t {
    void *data;
    word dimensions[];
}
```

The `data` field points to the array data, while the `dimensions` array contains the array dimensions. The run-time does not maintain or use the dimensions; they are provided for application use.

All remaining bits in the mobile type descriptor word are used to represent the inner type of the array. For example when declaring an array of bytes, a numeric data type descriptor will follow.

To give an illustrated example, a two-dimensional array of integers would be described by the following descriptor in figure 77.

B.2.3 2: Mobile Channel Bundle

An array of channel word or *channel bundle*. Each element of the mobile type maybe used as a communication channel.

The type flags define the following properties:

0 *shared*: the channel bundle is shared, this changes its communications semantics and allocates internal memory for semaphores making the lock and unlock operations valid. A shared channel bundle is copied on communication, where as unshared one is moved.

1 *pony*: allocate additional space for *pony* data [230]. This space is allocated after the channel words and its usage is defined by the compiler.

2 *unused*.

3-11 *channel count*: number of channels in the bundle.

All channel bundles are reference counted, including unshared bundles. Copies should be produced with the clone or process parameter copy operations and released with the release operation. In future, references to the same channel bundle may have different pointers and memory backing them. The channel operations will link the words in the separate bundles to provide seamless communication. For this reason it should not be assumed that the output of a clone, copy or move operation will be the same as its input.

B.2.4 3: Mobile Barrier

The type flags hold the numeric type of the barrier. The actual structure of the barrier is implementation specific, thus the barrier should simply be treated as a handle. Available barrier types are:

0 *full*: A full barrier used for common barrier synchronisation.

1 *forking*: A forking barrier, supports only a single process synchronising on the barrier. The barrier is automatically freed when synchronisation completes. Child processes are enrolled on the barrier when created and resign by freeing it on termination. The parent process synchronises to wait for completion of all children.

2 *mobile process*: This is specific to the mobile process implementation in the occ21 compiler and should not be used elsewhere.

B.2.5 4: Mobile Process

Reserved for future definition.

B.2.6 5: Mobile Type

Can be any mobile type. This is used for constructing containers for other mobile types, where the mobile type contain may not be known or may be variable. For example an array of mixed mobile types.

B.2.7 6: Mobile Type Descriptor

Reserved for future definition. Mobile type descriptors will be used to describe types which do not fit in a single machine word descriptor, as discussed in B.1.

B.2.8 7: Mobile Data

Mobile data of unspecified type. The inner type of the data is not declared. Thus it may not contain nested mobiles or undergo type conversion on communication boundaries. For implementation reasons the present compiler uses this type for implementing static mobiles of fixed size and storing the workspaces of recursive processes.

B.2.9 8: Fixed Array

Reserved for future definition.

B.2.10 9: Array Options

This special type defines additional options for a mobile array type. It should immediately follow the mobile array descriptor before the inner type. The type flag bits are used as follows:

0 *DMA*: data in the array will be used for DMA access by hardware. This hints to the runtime memory allocator that the memory backing the array must be accessible by peripheral busses in the system. It also triggers the allocation of an additional data pointer after the dimension elements to store the hardware address of the array data. The hardware address of the array data memory may different from that in the `data` pointer when the system uses address translation.

1 *separate*: data is separately allocated and should be released by the runtime. This allows the allocation of empty (zero-length) mobile arrays, which are later bound to other mobiles. When the mobile array is released the mobile type pointed to by the `data` pointer will also be released.

2 *unused*.

3-6 *alignment* for the array data as a power of two.

B.3 Operations

A number of operations are defined over mobile types, this section gives a brief overview of them.

B.3.1 malloc, mrelease

The *malloc* operation acts as in C, allocating a number of bytes of memory for use by the application. Internally this is a hot-path for allocating mobile data (type 7). Memory allocated via `malloc` is valid for use as a mobile type. `mrelease` acts like C's `free`; however, it is only defined for memory allocated by `malloc`.

For legacy reasons, in the present implementation allocations of size 0 will return `NULL` pointers, and the release of `NULL` pointers using mrelease is invalid.

B.3.2 mt_alloc, mt_release

These operations allocate and release mobile types. `mt_alloc` takes a type descriptor and a size parameter. For arrays the size parameter indicates the total number of elements, for channel bundles the channel count and for mobile data the number of bytes. Other types do not use the size parameter. As with `mrelease`, `mt_release` on a `NULL` pointer is not valid.

B.3.3 mt_clone

A clone operation is provided which can create a copy of any mobile type, including inner mobiles. For data types clone will copy the data into a new independent mobile type, for channel bundles and barriers the reference and enroll counts will simply be updated. With nested trees of mobile types the behaviour depends on the subtypes, for example copying an array of barriers will produce a new array, but the barriers referenced will be the same with the associated reference and enroll counts updated.

B.3.4 mt_in, mt_out

The full set of channel operations are defined for mobile types. Most operations take a pointer to a mobile type pointer. During output of a mobile type the source pointer will be set `NULL` in order to preserve reference integrity. The exception to this is when outputting shared channel bundles and barriers, in which case the reference count is updated and the source pointer left untouched.

B.3.5 mt_lock, mt_unlock

These operations take a mobile type and a lock type. The associated lock semaphore is manipulated, potentially causing the process to block if the lock is unavailable. These operations are only defined for shared channel bundles where they are used to lock associated directions of communication.

In future these operations maybe extended to shared data types.

B.3.6 mt_sync, mt_enroll, mt_resign

Barrier synchronisation, enroll and resign operations. Enroll and resign operations are typically implicit. Explicit operations are only required when handling groups of parallel processes. Under normal operation communicating a barrier automatically enrolls the receiver and releasing a barrier resigns the releasing process.

B.3.7 mt_bind

This special purpose operation binds a mobile to separate piece of memory or another mobile type. It is only supported for mobile arrays. Three types of bind are supported:

- *irtual*: bind the mobile to a virtual memory address. A physical address is stored for later reference if the associated array options structure is present and indicates DMA operation.
- *physical*: bind the mobile to a physical memory address, generating the virtual address for use in the application.
- *DMA*: make sure the memory in the mobile is hardware accessible, if not, reallocate the mobile into hardware accessible memory copying the data and releasing the original mobile type.

B.3.8 mt_resize

Resize a mobile type using the provided size parameter. At present only data resizing is supported and only for mobile arrays. This operation may be able to perform resizing without memory reallocation and copying, and also deals with releasing and initialising pointers for nested mobile types.

Bibliography

[1] ACM SIGPLAN Notices. ISSN 0362-1340.

[2] CCSP Comparison Benchmarks.
`http://projects.cs.kent.ac.uk/projects/kroc/svn/kroc/trunk/tests/ccsp-comparisons/`

[3] Complex Systems Modelling and Simulation infrastructure (CoSMoS).
`http://www.cosmos-research.org`

[4] DARPA High Productivity Computing Systems.
`http://www.highproductivity.org`

[5] GNU C Compiler.
`http://gcc.gnu.org`

[6] GNU Pth - The GNU Portable Threads.
`http://www.gnu.org/software/pth/`

[7] Kent Retargetable occam Compiler.
`http://projects.cs.kent.ac.uk/projects/kroc/trac/`

[8] LangPop.com - Programming Language Popularity.
`http://www.langpop.com`

[9] LLVM Language Reference Manual.
`http://www.llvm.org/docs/LangRef.html`

[10] LLVM Project.
`http://llvm.org`

[11] Ohloh Monthly Commits.
`http://www.ohloh.net/languages/compare`

[12] Surveyor Corporation SRV-1.
`http://www.surveyor.com/blackfin/`

[13] The Computer Language Benchmarks Game.
`http://shootout.alioth.debian.org/`

[14] Tock Compiler.
`http://projects.cs.kent.ac.uk/projects/tock/trac/`

[15] *IEEE Standard 1149.1-1990 Test Access Port and Boundary-Scan Architecture-Description*. IEEE, 1990.

[16] *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5. 415000.

[17] ISO/IEC TR 19768 – C++ Library Extensions, 2005.
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf`

[18] *The Open Group Base Specifications Issue 7: POSIX.1-2008*. The Open Group, December 2008.
`http://pubs.opengroup.org/onlinepubs/9699919799/`

[19] OpenMP Application Program Interface, Version 3.0, May 2008.

[20] ISO/IEC 14882:2011 – C++11, 2011.
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`

[21] *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5.

[22] Boost C++ Libraries, 2013.
`http://www.boost.org`

[23] Rust Language Homepage, February 2013.
`http://www.rust-lang.org`

[24] A. Acharya, M. Tambe, and A. Gupta. Implementation of production systems on message-passing computers. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):477–487, Jul 1992. ISSN 1045-9219.

[25] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[26] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu. The Tofu Interconnect. In *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, pages 87 –94, aug. 2011.

[27] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Loose Synchronization for Large-Scale Networked Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.

[28] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139:140, 2005.

[29] AMD. AMD Graphics Cores Next (GCN) Architecture. Technical report, AMD, 2012.
`http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf`

[30] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[31] Bruce Anderson. Type syntax in the language "C": an object lesson in syntactic innovation. *SIGPLAN Not.*, 15(3):21–27, March 1980. ISSN 0362-1340.
`http://doi.acm.org/10.1145/947626.947627`

[32] James P. Anderson, Samuel A. Hoffman, Joseph Shifman, and Robert J. Williams. D825 - a multiple-computer system for command & control. In *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*, AFIPS '62 (Fall), pages 86–96, New York, NY, USA, 1962. ACM.
`http://doi.acm.org/10.1145/1461518.1461527`

[33] Kenneth R. Anderson and Duane Rettig. Performing Lisp analysis of the FANNKUCH benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, 1994. ISSN 1045-3563.

[34] Paul S. Andrews, Fiona Polack, Adam T. Sampson, Jon Timmis, Lisa Scott, and Mark Coles. Simulating biology: towards understanding what the simulation shows. In Susan Stepney, Fiona Polack, and Peter Welch, editors, *Proceedings of the 2008 Workshop on Complex Systems Modelling and Simulation, York, UK, September 2008*, pages 93–123. Luniver Press, 2008.
`http://www.cosmos-research.org/docs/cosmos2008-understanding.pdf`

[35] Paul S. Andrews, Adam T. Sampson, John Markus Bjørndalen, Susan Stepney, Jon Timmis, Douglas N. Warren, and Peter H. Welch. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24. MIT Press, Cambridge, MA, August 2008.
`http://www.cosmos-research.org/docs/alife2008-space.pdf`

[36] ARM. ARMv7-M Architecture Reference Manual, February 2010.

[37] Joe Armstrong. Concurrency oriented programming in Erlang. Invited talk, Frühjahrsfachgespräch (FFG), 2003.

[38] Joe Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7.
`http://doi.acm.org/10.1145/1238844.1238850`

[39] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. In *Proceedings of the IEEE*, volume 93, pages 449–466, Feb 2005.

[40] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, 1994. ISSN 0004-5411.

[41] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, May 1960. ISSN 0001-0782.
`http://doi.acm.org/10.1145/367236.367262`

[42] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM.
`http://doi.acm.org/10.1145/1455567.1455599`

[43] John Backus. History of programming languages I. chapter The history of Fortran I, II, and III, pages 25–74. ACM, New York, NY, USA, 1981. ISBN 0-12-745040-8.
`http://doi.acm.org/10.1145/800025.1198345`

[44] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, August 1977. ISSN 0362-1340.
`http://doi.acm.org/10.1145/872734.806932`

[45] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211 –243, February 1993. ISSN 0018-9219.

[46] Fred 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 Series*, pages 269–288, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN 1-58603-381-6.
`http://www.cs.kent.ac.uk/pubs/2003/1721`

[47] Frederick R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Alan Chalmers, Majid Mirmehdi, 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. ISBN 1 58603 202 X.
`http://www.cs.kent.ac.uk/pubs/2001/1296`

[48] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent at Canterbury, June 2003.
`http://www.cs.kent.ac.uk/pubs/2003/1701`

[49] Frederick R.M. Barnes and Carl G. Ritson. Process Oriented Device Driver Development. *Concurrency and Computation: Practice and Experience*, 2008. Submitted for special edition.

[50] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller,

editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN 1-58603-202-X.

[51] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN 1-58603-268-2.

[52] Geoff Barrett, Michael Goldsmith, Geraint Jones, and A. Kay. The meaning and implementation of PRI ALT in occam. In Charlie Askew, editor, *OUG-9: Occam and the Transputer – Research and Applications*, pages 37–46, September 1988. ISBN 90 5199 010 3.

[53] Iann M Barron. The Transputer. In *MiniMicro West 83, San Francisco, CA*, volume 2(5), pages 1–8. Electric Conventions Management, November 1983.

[54] L.A. Barroso and U. Holzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33 –37, dec. 2007. ISSN 0018-9162.

[55] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004. ISSN 0164-0925.

[56] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The Essence of Data Access in C. In AndrewP. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-27992-1.
`http://dx.doi.org/10.1007/11531142_13`

[57] Robert D Blumofe. *Executing multithreaded programs efficiently*. PhD thesis, Massachusetts Institute of Technology, 1995.

[58] Robert D. Blumofe, Christopher F. Jöerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-701-6.

[59] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. ISSN 0004-5411.

[60] Neil C. C. Brown. *Communicating Haskell Processes*. PhD thesis, University of Kent at Canterbury, 2011.

[61] T. Brunklaus and Leif Kornstaedt. A Virtual Machine for Multi-language Execution. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, 2002.

[62] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[63] Wentong Cai and Stephen J. Turner. An Approach to the Run-Time Monitoring of Parallel Programs. *The Computer Journal*, 37(4):333–345, March 1994.
`citeseer.ist.psu.edu/cai94approach.html`

[64] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04*, pages 52–60, 2004.

[65] Juanito Camilleri. An operational semantics for occam. *International Journal of Parallel Programming*, 18:365–400, 1989. ISSN 0885-7458.
`http://dx.doi.org/10.1007/BF01379186`

[66] Luiz Fernando Capretz. A brief history of the object-oriented approach. *SIGSOFT Softw. Eng. Notes*, 28(2):6, March 2003. ISSN 0163-5948.
`http://doi.acm.org/10.1145/638750.638778`

[67] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
`http://hpc.sagepub.com/content/21/3/291.abstract`

[68] Bradford L. Chamberlain. A Brief Overview of Chapel, January 2013.
`http://chapel.cray.com/papers/BriefOverviewChapel.pdf`

[69] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* The MIT Press, 2007. ISBN 0262533022, 9780262533027.

[70] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005. ISSN 0362-1340.
`http://doi.acm.org/10.1145/1103845.1094852`

[71] A. Cimitile, U. De Carlini, and U. Villano. Replay-based debugging of occam programs. *Software Testings, Verfication and Reliability*, 3(2):83–100, 1993.

[72] Rance Cleaveland, Gerald Lüttgen, and V. Natarajan. Priority in process algebras. Technical report, Institute for Computer Applications in Science and Engineering, 1999.

[73] Cliff Click. Azul's experiences with hardware transactional memory. In *HP Labs – Bay Area Workshop on Transactional Memory*, January 2009.

[74] William D Clinger. Foundations of Actor Semantics. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[75] W. F. Clocksin and Christopher S. Mellish. *Programming in Prolog (3rd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987. ISBN 0-387-17539-3.

[76] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel C. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9.
`http://doi.acm.org/10.1145/1065944.1065950`

[77] William W Collier. *Reasoning About Parallel Architectures*. Prentice Hall, 1992. ISBN 0137671873.

[78] United States. Warren Commission. *Report of the President's Commission on the Assassination of President John F. Kennedy*, volume 36. US Govt. Print. Off., 1964.

[79] Guojing Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and Tong Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 536–545, September 2008. ISSN 0190-3918.

[80] HyperTransport Consortium. HyperTransport I/O Technology Overview, June 2004.
`http://www.hypertransport.org/docs/wp/HT_Overview.pdf`

[81] UPC Consortium et al. UPC Language Specifications Tech Report LBNL–59208. Technical report, Lawrence Berkeley National Lab, 2005.

[82] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963. ISSN 0001-0782.
`http://doi.acm.org/10.1145/366663.366704`

[83] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.
`http://doi.acm.org/10.1145/1463822.1463838`

[84] Cray. Cray XE6 Brochure, 2010.

[85] A. J. Critcklow. Generalized multiprocessing and multiprogramming systems. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 107–126, New York, NY, USA, 1963. ACM.
`http://doi.acm.org/10.1145/1463822.1463836`

[86] Ole-Johan Dahl. *SIMULA 67 common base language, (Norwegian Computing Center. Publication)*. 1968. ISBN B0007JZ9J6.

[87] Jim Davies and Steve Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243 – 271, 1995. ISSN 0304-3975.
`http://www.sciencedirect.com/science/article/pii/030439759400169J`

[88] K. Debattista, K. Vella, and J. Cordina. Wait-free cache-affinity thread scheduling. *IEE Proceedings Software*, 150(2):137–146, April 2003. ISSN 1462-5970.

[89] Department of Energy and Climate Change. Quarterly Energy Prices, December 2012. ISSN 1755-9103.
`http://www.decc.gov.uk/assets/decc/11/stats/publications/qep/7341-quarterly-energy-prices-december-2012.pdf`

[90] Edsger W. Dijkstra. Cooperating Sequential Processes. Technical report, Technological University, Eindhoven, The Netherlands, September 1965.

[91] Edsger W. Dijkstra. *Cooperating Sequential Processes*, pages 43–112. Academic Press, New York, NY, USA. , 1968.

[92] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968. ISSN 0001-0782.

[93] Edsger W. Dijkstra. How do we tell truths that might hurt? *SIGPLAN Not.*, 17(5):13–15, 1982.

[94] Edsger W. Dijkstra. The origin of concurrent programming. chapter Cooperating sequential processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002. ISBN 0-387-95401-5.
`http://dl.acm.org/citation.cfm?id=762971.762974`

[95] Damian Dimmich. *A Process Oriented Approach to Solving Problems of Parallel Decomposition and Distribution.* PhD thesis, Computing, University of Kent, CT2 7NF, June 2009.
`http://www.cs.kent.ac.uk/pubs/2009/3058`

[96] 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. ISBN 1-58603-561-4.
`http://www.cs.kent.ac.uk/pubs/2005/2254`

[97] Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Trans. Archit. Code Optim.*, 9(4):36:1–36:25, January 2013. ISSN 1544-3566.
`http://doi.acm.org/10.1145/2400682.2400695`

[98] Ulrich Drepper. ELF Handling for Thread-Local Storage. Technical report, Red Hat Inc, December 2005. Version 0.20.
`http://www.akkadia.org/drepper/tls.pdf`

[99] Ulrich Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat Inc, November 2007. Version 1.00.
`http://www.akkadia.org/drepper/cpumemory.pdf`

[100] R. Kent Dybvig. *The Scheme Programming Language, 4th Edition.* The MIT Press, 4th edition, 2009. ISBN 026251298X, 9780262512985.

[101] A. Einstein and F.A. Davis. *The principle of relativity*. Dover Publications, 1952.

[102] Steven Ericsson-Zenith. *Process Interaction Models*. PhD thesis, Universite Pierre et Marie Curie, PARIS (VI), 1992.

[103] Chris Exton and Michael Kölling. Concurrency, objects and visualisation. In *ACSE '00: Proceedings of the Australasian conference on Computing education*, pages 109–115, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-271-9.

[104] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *SIGPLAN Workshop on Transactional Computing*. ACM, August 2007.

[105] Michael B. Feldman. Who's Using Ada? Real-World Projects Powered by the Ada Programming Language, February 2013.
`http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html`

[106] Georgina Ferry. *A Computer Called LEO: Lyons Tea Shops and the world's first office computer*. Harper Perennial, 2004. ISBN 1841151866.

[107] C. J. Fidge. A formal definition of priority in CSP. *ACM Trans. Program. Lang. Syst.*, 15(4):681–705, September 1993. ISSN 0164-0925.
`http://doi.acm.org/10.1145/155183.155221`

[108] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *DAMP '07*, pages 37–44, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5.

[109] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948 –960, September 1972. ISSN 0018-9340.

[110] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston,

MA, USA, 1995. ISBN 0201575949.
`http://wotug.org/parallel/books/addison-wesley/dbpp/text/node1.html`

[111] Cédric Fournet and Georges Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-44044-4.
`http://dx.doi.org/10.1007/3-540-45699-6_6`

[112] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[113] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. ISSN 0164-0925.

[114] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35:97–107, February 1992. ISSN 0001-0782.
`http://doi.acm.org/10.1145/129630.129635`

[115] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[116] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162, 2006. ISSN 0163-5964.

[117] James Gosling. *The Java language specification*. Prentice Hall, 2000.

[118] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60 –69, june 1990. ISSN 0018-9162.

[119] Per Brinch Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, July 1972. ISSN 0001-0782.
`http://doi.acm.org/10.1145/361454.361473`

[120] Per Brinch Hansen. Monitors and concurrent Pascal: a personal history. *SIGPLAN Not.*, 28(3):1–35, March 1993. ISSN 0362-1340.
`http://doi.acm.org/10.1145/155360.155361`

[121] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan Gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, March 2012. ISSN 0272-1732.
`http://dx.doi.org/10.1109/MM.2011.108`

[122] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003. ISSN 0362-1340.

[123] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: ACM SIGPLAN workshop on Haskell*, pages 49–61, New York, NY, USA, 2005. ACM. ISBN 1-59593-071-X.

[124] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. *SIGPLAN Not.*, 42(9):251–264, October 2007. ISSN 0362-1340.
`http://doi.acm.org/10.1145/1291220.1291192`

[125] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 293–298, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3.
`http://doi.acm.org/10.1145/800055.802046`

[126] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems*, pages 522 – 529. IEEE Computer Society, May 2003. ISSN 1063-6927.

[127] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. ISSN 0164-0925.

[128] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993. ISSN 0164-0925.
`http://doi.acm.org/10.1145/161468.161469`

[129] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993. ISSN 0164-0925.

[130] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
`http://dl.acm.org/citation.cfm?id=1624775.1624804`

[131] Rich Hickey. The Clojure Programming Language. In *DLS '08: Proceedings of the 2008 Symposium on Dynamic Languages*, pages 1–1, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2.

[132] I. Hickson and A. Melnikov. RFC 6455: The WebSocket Protocol, September 2011. ISSN 2070-1721.
`http://tools.ietf.org/html/rfc6455`

[133] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974. ISSN 0001-0782.
`http://doi.acm.org/10.1145/355620.361161`

[134] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[135] Guido Hogen, Andrea Kindler, and Rita Loogen. Automatic Parallelization of Lazy Functional Programs. In *Proc. of 4th European Symposium on Programming, ESOP'92, LNCS 582:254-268*, pages 254–268. Springer-Verlag, 1992.

[136] Tim Hoverd and Adam T. Sampson. A Transactional Architecture for Simulation. In *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '10, pages 286–290, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4015-3.
`http://dx.doi.org/10.1109/ICECCS.2010.7`

[137] Jean D. Ichbiah, John G.P. Barnes, Robert J. Firth, and Mike Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, 1991. ISBN 0-521-39267-5.

[138] INMOS. At Last–The Transputer Unveiled. *Electronic Engineering*, 55:11, December 1983.

[139] INMOS Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[140] Intel. Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor. Technical report, Intel, March 2004.

[141] Intel. An Introduction to the Intel QuickPath Interconnect, January 2009.

[142] Intel. *Intel® Architecture Instruction Set Extensions Programming Reference*, February 2012.

[143] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, January 2013.

[144] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2013.

[145] International Business Machines Corporation. Power ISA, Version 2.05, October 2007.

[146] Christian L. Jacobsen. *A Portable Runtime for Concurrency Research and Application*. PhD thesis, Computing Laboratory, University of Kent, December 2006.

[147] 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, September 2004. ISBN 1 58603 458 8.
`http://www.cs.kent.ac.uk/pubs/2004/2004`

[148] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2012. ISBN 1420082795, 9781420082791.

[149] Alan Kay. Dr. Alan Kay on the Meaning of "Object-Oriented Programming", July 2003.
`http://www.purl.org/stefan_ram/pub/doc_kay_oop_en`

[150] Alan C. Kay. The early history of Smalltalk. *SIGPLAN Not.*, 28(3):69–95, March 1993. ISSN 0362-1340.
`http://doi.acm.org/10.1145/155360.155364`

[151] Gabriel Kerneis and Juliusz Chroboczek. Continuation-Passing C, compiling threads to events through continuations. *Higher-Order and Symbolic Computation*, 24:239–279, 2011. ISSN 1388-3690.
`http://dx.doi.org/10.1007/s10990-012-9084-5`

[152] Donald E. Knuth. Structured Programming with go to Statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974. ISSN 0360-0300.
`http://doi.acm.org/10.1145/356635.356640`

[153] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, February 2012. ISSN 0362-1340.
`http://doi.acm.org/10.1145/2370036.2145835`

[154] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. Technical report, Analytics Press, August 2011.

[155] Leif Kornstaedt. Alice in the Land of Oz – An Interoperability-based Implementation of a Functional Language on Top of a Relational Language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Computer Science*, volume 59, Firenze, Italy, September 2001. Elsevier Science Publishers.

[156] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, 2008. ISSN 0362-1340.

[157] Leslie Lamport. A New Approach to Proving the Correctness of Multiprocess Programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, January 1979. ISSN 0164-0925.
`http://doi.acm.org/10.1145/357062.357068`

[158] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[159] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979. ISSN 0163-5980.

[160] R. Greg Lavender and Douglas C. Schmidt. Pattern languages of program design 2. chapter Active object: an object behavioral pattern for concurrent programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. ISBN 0-201-895277.
`http://dl.acm.org/citation.cfm?id=231958.232967`

[161] Doug Lea. A Java Fork/Join Framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3.

[162] T.J. Leblanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *Computers, IEEE Transactions on*, C-36(4):471–482, April 1987. ISSN 0018-9340.

[163] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, June 1988. ISSN 0362-1340.
`http://doi.acm.org/10.1145/960116.54016`

[164] Gavin Lowe. Extending CSP with Tests for Availability. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, G. S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 325–347, November 2009. ISBN 978-1-60750-065-0.

[165] Gavin Lowe. Implementing Generalised Alt. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 1–34, June 2011. ISBN 978-1-60750-773-4.

[166] Honghui Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95*, pages 37–37, 1995.

[167] Zoltan Majo and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0773-4.
`http://doi.acm.org/10.1145/1987816.1987832`

[168] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

ISBN 1-58113-830-X.
`http://doi.acm.org/10.1145/1040305.1040336`

[169] Luc Maranget, Susmit Sarkar, and Peter Sewell. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. Technical report, University of Cambridge, October 2012.
`http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`

[170] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *ICFP '09: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming*, August 2009.
`http://community.haskell.org/~simonmar/papers/multicore-ghc.pdf`

[171] James Martin. *Application Development without Programmers*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1982. ISBN 0130389439.

[172] David May. OCCAM. *SIGPLAN Not.*, 18(4):69–79, April 1983.

[173] David May. Communicating Process Architecture for Multicores. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 21–32, jul 2007. ISBN 978-1586037673.

[174] David May. The XMOS XS1 Architecture, 2009.
`https://www.xmos.com/en/download/public/The-XMOS-XS1-Architecture(1.0).pdf`

[175] John May and Francine Berman. Panorama: a portable, extensible parallel debugger. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 96–106, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-633-6.

[176] M. D. May, P. W. Thompson, and Peter H. Welch. *Networks, Routers and Transputers*. IOS Press, Amsterdam, 1993.

[177] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. `http://doi.acm.org/10.1145/367177.367199`

[178] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989. ISSN 0360-0300.

[179] Michael B. Mensky. *Quantum Measurements and Decoherence*, chapter 4.1.1, page 315. Springer. ISBN 0-7923-6227-6, 2000.

[180] Rick Merritt. CPU designers debate multi-core future, June 2008. `http://eetimes.com/electronics-news/4076123/CPU-designers-debate-multi-core-future`

[181] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2. `http://doi.acm.org/10.1145/248052.248106`

[182] Maged M. Michael and Michael L. Scott. Non-blocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, 51:1–26, 1998.

[183] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353.

[184] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN 0521658691.

[185] 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.

[186] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[187] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000. ISSN 0360-0300.

[188] Andrew M. Mironov. Theory of processes. *CoRR*, abs/1009.2259, 2010.

[189] David A. P. Mitchell, Jonathan A. Thompson, Gordon A. Manson, and Graham R. Brookes. *Inside The Transputer*. Blackwell Scientific Publications, Ltd., Oxford, UK, 1990. ISBN 0-632-01689-2.

[190] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter*, 11(5):33 –35, September 2006. ISSN 1098-4232.

[191] James Moores. *The Design and Implementation of OCCAM/CSP Support for a Range of Languages and Platforms*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, April 2002.
`http://www.cs.kent.ac.uk/pubs/2002/1843`

[192] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1.
`http://doi.acm.org/10.1145/1254810.1254820`

[193] Microsoft Developer Network. Fibers (Windows), February 2013.
`http://msdn.microsoft.com/en-us/library/ms682661.aspx`

[194] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, NVIDIA, 2009.

http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[195] Martin Odersky and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2009.

[196] C Olsen and L Morrow. Multi-processor computer system having low power consumption. *Power-Aware Computer Systems*, pages 97–100, 2003.

[197] C. O'Neil. The TDS occam 2 debugging system. In Traian Muntean, editor, *OUG-7: Parallel Programming of Transputer Based Machines*, pages 9–14, sep 1987. ISBN 90 5199 0073.

[198] Oracle. JDK 1.1 for Solaris Developer's Guide - Multithreading Models, 2010. http://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqk/index.html#ch2mt-41

[199] Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03358-2.
http://dx.doi.org/10.1007/978-3-642-03359-9_27

[200] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. The HiPE/x86 Erlang Compiler. *Functional and Logic Programming*, 2441/2002:228–244, January 2002.
http://citeseer.ist.psu.edu/sagonas02hipex.html

[201] Rob Pike. Newsqueak: A language for communicating with mice. Technical report, Bell Labs, 1989.

[202] Rob Pike. Another Go at Language Design. In *Stanford University Computer Systems Laboratory Colloquium*, 2010.

[203] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[204] Michael D. Poole. occam-for-all – Two Approaches to Retargeting the INMOS occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments – Proceedings of WoTUG 19*, pages 167–178, Nottingham-Trent University, UK, March 1996. World occam and Transputer User Group, IOS Press, Netherlands. ISBN 90-5199-261-0.
`http://www.cs.kent.ac.uk/pubs/1996/285`

[205] Michael D. Poole. Extended Transputer Code – a Target-Independent Representation of Parallel Programs. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 187–198, March 1998. ISBN 90 5199 391 9.

[206] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000. ISSN 1096-9128.
`http://dx.doi.org/10.1002/1096-9128(200005)12:6<445::AID-CPE484>3.0.CO;2-A`

[207] U. Ramachandran, M. Solomon, and M. Vernon. Hardware support for interprocess communication. In *Proceedings of the 14th annual international symposium on Computer architecture*, ISCA '87, pages 178–188, New York, NY, USA, 1987. ACM. ISBN 0-8186-0776-9.
`http://doi.acm.org/10.1145/30350.30371`

[208] B. Randell. Software engineering in 1968. In *Proceedings of the 4th international conference on Software engineering*, ICSE '79, pages 1–10, Piscataway, NJ, USA, 1979. IEEE Press.
`http://dl.acm.org/citation.cfm?id=800091.802915`

[209] James Reinders. *Intel Threading Building Blocks*. O'Reilly Media Inc, Sebastopol, CA, 2007.

[210] Martin Reiser. *The Oberon system: user guide and programmer's manual*. ACM, New York, NY, USA, 1991. ISBN 0-201-54422-9.

[211] John Reppy, Claudio V. Russo, and Yingqi Xiao. Parallel concurrent ML. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 257–268, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

[212] John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-48089-2.

[213] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM. ISBN 0-89791-227-6.

[214] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993. ISSN 0892-4635.

[215] Dennis M. Ritchie. The development of the C language. *SIGPLAN Not.*, 28(3):201–208, March 1993. ISSN 0362-1340.
`http://doi.acm.org/10.1145/155360.155580`

[216] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974. ISSN 0001-0782.
`http://doi.acm.org/10.1145/361011.361061`

[217] Carl G. Ritson. Translating ETC to LLVM Assembly. In Peter H. Welch, editor, *Communicating Process Architectures 2009*, volume 67 of *Concurrent Systems Engineering Series*, pages 145–158, Amsterdam, The Netherlands, November 2009. IOS Press.

[218] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and

Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009. ISBN 978-3-642-02052-0.
`http://www.cs.kent.ac.uk/pubs/2009/2928`

[219] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Video Processing in occam-pi. In P.H. Welch, J. Kerridge, and F.R.M. Barnes, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pages 311–329, Amsterdam, The Netherlands, September 2006. IOS Press. ISBN 1-58603-671-8.
`http://www.cs.kent.ac.uk/pubs/2006/2494`

[220] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727 – 740, 2012. ISSN 0167-6423.
`http://www.sciencedirect.com/science/article/pii/S0167642311001286`

[221] Carl G. Ritson and Jonathan Simpson. Virtual Machine Based Debugging for occam-pi. In Peter H. Welch, editor, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 293–307, Amsterdam, The Netherlands, September 2008. IOS Press.

[222] Carl G. Ritson and Peter H. Welch. A Process-Oriented Architecture for Complex System Modelling. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 249–266, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN 978-1-58603-767-3.
`http://www.cs.kent.ac.uk/pubs/2007/2716`

[223] Carl G. Ritson and Peter H. Welch. A Process-Oriented Architecture for Complex System Modelling. *Concurrency and Computation: Practice and Experience*, 22:965–980, March 2010.
`http://www.cs.kent.ac.uk/pubs/2010/3066`

[224] Phil Rogers. The Programmer's Guide to the APU Galaxy, June 2011.
`http://amddevcentral.com/afds//assets/keynotes/Phil%20Rogers%20Keynote-FINAL.pdf`

[225] A. W. Roscoe. Denotational Semantics for occam. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 306–329, London, UK, UK, 1985. Springer-Verlag. ISBN 3-540-15670-4.
`http://dl.acm.org/citation.cfm?id=646723.702725`

[226] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. ISBN 0136744095.

[227] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, Munich, Germany. ISBN 1-84150144-1, February 2006.

[228] Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.
`http://offog.org/publications/ats-thesis.pdf`

[229] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, Institut d'Informatique Fondamentale, 2005.

[230] Mario Schweigler. *A Unified Model for Inter- and Intra-processor Concurrency*. PhD thesis, University of Kent, Canterbury, Kent, CT2 7NF, United Kingdom, August 2006.
`http://www.cs.kent.ac.uk/pubs/2006/2429`

[231] Jan Schwinghammer. A Concurrent Lambda-Calculus with Promises and Futures. Master's thesis, Programming Systems Lab, Universität des Saarlandes, February 2002.

[232] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3.
`http://doi.acm.org/10.1145/224964.224987`

[233] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3.
`http://doi.acm.org/10.1145/1375527.1375568`

[234] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51 – 92, 1993. ISSN 0004-3702.
`http://www.sciencedirect.com/science/article/pii/0004370293900349`

[235] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. Mobile Robot Control: The Subsumption Architecture and occam-pi. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 225–236, Amsterdam, The Netherlands, September 2006. IOS Press. ISBN 1-58603-671-8.
`http://www.transterpreter.org/wiki/Publications`

[236] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference (Vol. 1): Volume 1-The MPI Core*, volume 1. MIT Press, 1998.

[237] D. C. Stanga. UNIVAC 1108 Multiprocessor System. *Managing Requirements Knowledge, International Workshop on*, 0:67, 1967.

[238] R. Steigerwald and M. Nelson. Concurrent programming in Smalltalk-80. *SIGPLAN Not.*, 25:27–36, August 1990. ISSN 0362-1340.
`http://doi.acm.org/10.1145/87416.87427`

[239] Susan Stepney. GRAIL: Graphical Representation of Activity, Interconnection and Loading. In Traian Muntean, editor, *7th Technical meeting of the occam User Group,Grenoble, France*. IOS Amsterdam, 1987.

[240] Susan Stepney. Understanding Multi-transputer Execution. In *IT UK 88, University College Swansea, UK*, 1988.

[241] Michael Süß and Claudia Leopold. Common Mistakes in OpenMP and How to Avoid Them. In MatthiasS. Mueller, BarbaraM. Chapman, BronisR. Supinski, AllenD. Malony, and Michael Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 312–323. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-68554-8.
`http://dx.doi.org/10.1007/978-3-540-68555-5_26`

[242] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.
`http://www.gotw.ca/publications/concurrency-ddj.htm`

[243] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Professional Technical Reference, 5th edition, October 2010. ISBN 0132126958.

[244] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, January 2006. ISBN 0131429388.
`http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0131429388`

[245] K. Thompson and D.M. Ritchie. *UNIX Programmer's Manual*. Bell Telephone Laboratories, 1978.

[246] TIOBE Software. TIOBE Programming Community Index for January 2013, Jan 2013.
`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[247] UNFPA. State of World Population 2012. Technical report, United Nations, 2012. ISBN 978-1-61800-009-5.
`http://www.unfpa.org/public/home/publications/pid/12511`

[248] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. ISSN 0001-0782.

[249] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3.
`http://doi.acm.org/10.1145/224964.224988`

[250] Kevin Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, University of Kent, December 1998.
`http://www.cs.kent.ac.uk/pubs/1998/928`

[251] Mitchell Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, LFP '80, pages 19–28, New York, NY, USA, 1980. ACM.
`http://doi.acm.org/10.1145/800087.802786`

[252] Douglas Watt. Programming XC on XMOS devices, September 2009.
`https://www.xmos.com/en/download/public/Programming-XC-on-XMOS-Devices(X9577A).pdf`

[253] D Weaver. OpenSPARC Internals. Technical report, Sun Microsystems, 2008.

[254] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual: Version 9*. Prentice-Hall, 1994.

[255] Peter H. Welch and Frederick 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. ISBN 1-58603-561-4.
`http://www.cs.kent.ac.uk/pubs/2005/2272`

[256] Peter H. Welch, Frederick R. M. Barnes, and Fiona A. C. Polack. Communicating Complex Systems. In *ICECCS '06: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 107–120, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2530-X.

[257] Peter H. Welch and Frederick R.M. Barnes. Communicating Mobile Processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005. ISBN 3-540-25813-2.
`http://www.cs.kent.ac.uk/pubs/2005/2162`

[258] Peter H. Welch, Neil C. C. Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Alting Barriers: Synchronisation with Choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22:1049–1062, March 2010.
`http://www.cs.kent.ac.uk/pubs/2010/3068`

[259] Peter H. Welch, George R.R. Justo, and Colin J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press. ISBN 90-5199-140-1.
`http://www.cs.kent.ac.uk/pubs/1993/279`

[260] Peter H. Welch and Jan B. Pedersen. Santa Claus: Formal analysis of a process-oriented solution. *ACM Trans. Program. Lang. Syst.*, 32(4):14:1–14:37, April 2010.

ISSN 0164-0925.
http://doi.acm.org/10.1145/1734206.1734211

[261] Peter H. Welch and David C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments – Proceedings of WoTUG 19*, pages 143–166, Nottingham-Trent University, UK, March 1996. World occam and Transputer User Group, IOS Press, Netherlands. ISBN 90-5199-261-0.
http://www.cs.kent.ac.uk/pubs/1996/283

[262] P.H. Welch. Java Threads in the Light of occam/CSP. 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 Series*, pages 259–284, Amsterdam, April 1998. WoTUG, IOS Press. ISBN 90 5199 391 9.
http://www.cs.kent.ac.uk/pubs/1998/702

[263] Colin J. Willcock. *A Parallel X Window Server*. PhD thesis, University of Kent at Canterbury, 1992.

[264] N. Wirth. A plea for lean software. *Computer*, 28(2):64 –68, feb 1995. ISSN 0018-9162.

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

[266] David C. Wood and Frederick R. M. Barnes. Post-Mortem Debugging in KRoC. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 179–192, sep 2000. ISBN 1 58603 077 9.

[267] K. Zhang and G. Marwaha. Visputer–A Graphical Visualization Tool for Parallel Programming. *The Computer Journal*, 38(8):658–669, 1995.

Index